Technical Report ITL-96-1
May 1996

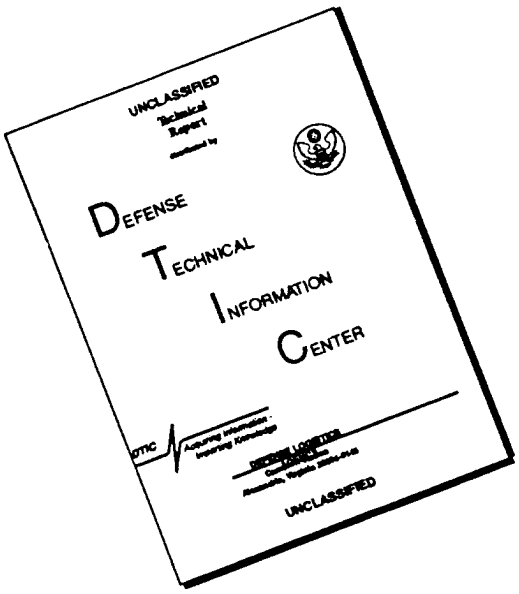
US Army Corps of Engineers
Waterways Experiment Station

Object Database Systems: A Tutorial

by Kofi Apenyo

Prepared for Headquarters, U.S. Army Corps of Engineers

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

Technical Report ITL-96-1
May 1996

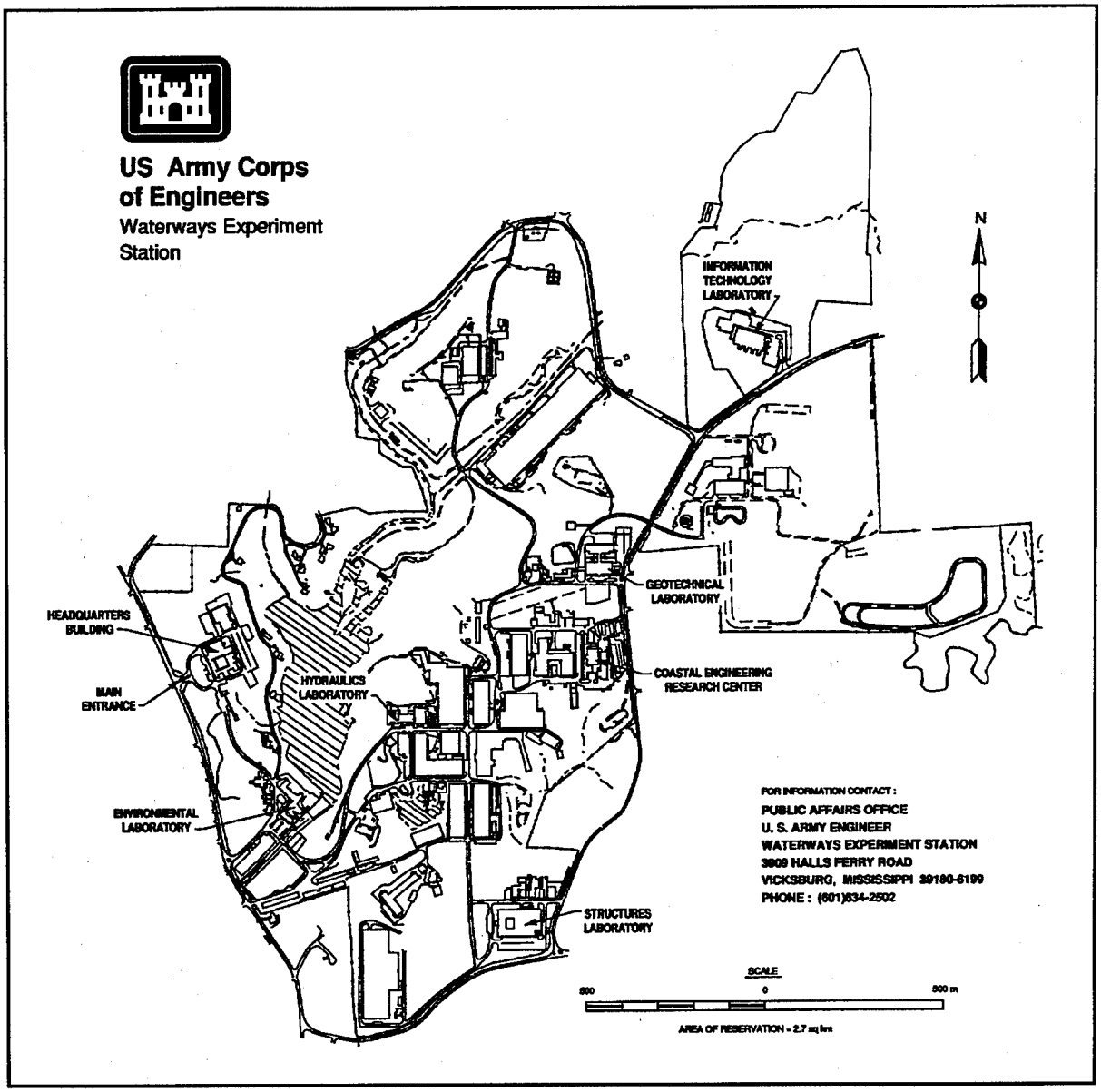
Object Database Systems: A Tutorial

by Kofi Apenyo

U.S. Army Corps of Engineers
Waterways Experiment Station
3909 Halls Ferry Road
Vicksburg, MS 39180-6199

Final report

Prepared for U.S. Army Corps of Engineers
Washington, DC 20314-1000



Waterways Experiment Station Cataloging-in-Publication Data

Apenyo, Kofi.
Object database systems : a tutorial / by Kofi Apenyo ; prepared for U.S. Army Corps of Engineers.
126 p. : ill. ; 28 cm. — (Technical report ; ITL-96-1)
Includes bibliographic references.
1. Object-oriented programming (Computer science) 2. Object-oriented databases. 3. CAD/CAM systems. I. United States. Army. Corps of Engineers. II. U.S. Army Engineer Waterways Experiment Station. III. Information Technology Laboratory (U.S. Army Engineer Waterways Experiment Station) IV. Title. V. Series: Technical report (U.S. Army Engineer Waterways Experiment Station) ; ITL-96-1.
TA7 W34 no.ITL-96-1

Contents

List of Figures

List of Tables

Preface

This report presents research performed by Dr. Kofi Apenyo of Jackson State University, Jackson, MS, under the supervision of Dr. Windell Ingram, Chief, Computer Science Division, Information Technology Laboratory (ITL), U.S. Army Engineer Waterways Experiment Station (WES) pursuant to an assignment agreement between Jackson State University and WES for the period 16 May 1995 to 11 August 1995. Dr. N. Radhakrishnan was Director of ITL during preparation of this report. The primary task was to research and introduce the emerging area of object database technology to selected ITL computer scientists. This tutorial is the result of that effort. The tutorial was written and, for ready comprehension, illustrated with numerous examples from the familiar object-oriented Microsoft Windows environment. The tutorial was presented in a 6-hour instructor-led course which was spread over 4 days to accommodate course assignments. The lectures were accompanied by live computer demonstrations on the Gemstone object database management system (DBMS) at Jackson State University. Gemstone was made available to course participants, who were encouraged to implement their assignments on the object DBMS.

During the preparation and publication of this report, Director of WES was Dr. Robert W. Whalin. Commander was COL Bruce K. Howard, EN.

1. INTRODUCTION

1.1 Introduction

The object-oriented (OO) paradigm was first introduced in the design of advanced programming languages in environments such as Smalltalk (Goldberg and Robson 1983). This occurred at a time when the relational approach had a stranglehold on data management. Even at its zenith it was becoming increasingly clear that relational systems could not handle certain problems properly. These problems were to be found in complex business modeling and in the technology areas such as computer-integrated manufacturing, information retrieval, CAD/CAM, CASE, and GIS. Elsewhere programming experience had shown that OO concepts would readily solve many of these problems. However, lack of persistence is a major drawback for using a programming language as the sole weapon for solving large problems. Accordingly, some of the more successful, well-established RDBMSs proceeded to make extensions to include the OO paradigm. Meanwhile, following the lead of Gemstone, database products such as Itasca, O_2, Objectivity/DB, ObjectStore, ONTOS, Poet, STATICE, and Versant took the approach of designing an architecture to directly support the OO paradigm. This pure OO approach to data management is the topic of this tutorial.

The OO paradigm is based on five fundamental concepts (Bertino and Martino 1991):

1) Each real-world entity is modeled by an object. Each object is associated with a unique identifier.

2) Each object has a set of instance variables and methods; the value of an instance variable can be an object or a set of objects. This characteristic permits arbitrary complex objects to be defined as an aggregation of other objects. The set of instance variables of an object and the set of methods represent the object structure and behavior, respectively.

3) The instance variable values represent the object's state. This state is accessed or modified by sending messages to the object to invoke the corresponding methods.

4) Objects sharing the same structure and behavior are grouped into classes. A class represents a template for a set of similar objects. Each object is an instance of some class.

5) A class can be defined as a specialization of one or more classes. A class defined as a specialization is called a subclass and inherits instance variables and methods from its superclass(es).

The OO approach takes a fundamental departure from traditional data management. It not only proposes a novel notion of entity, the data management metaphor for the chemist's molecule, but it permeates numerous technical data management areas and, indeed, the very philosophy of the data processing industry. For example, traditional data management is confined to the centralization of data and access to that data is handled by application programs, a separate function. By contrast, an OO DBA's primary task is not limited to modeling and design only, but includes programming as well. Thus some of the programmer's task is assumed by the DBA in an OO database system, thereby simplifying the programmer's task. Another simplifying feature of application programming is that OO systems provide code reusability. Reusability, a prize of OO systems, is achieved through inheritance and serves to reduce the amount of programming. Thirdly, the OO paradigm avoids the impedance mismatch that exists between data and program in traditional data management by advocating a rapprochement between the database and the programming language to yield a seamless application interface. In addition to increased programmer productivity, performance is greatly enhanced over comparable scenarios in relational systems. However, a price is exacted for these benefits in the increased complexity of the DBA's modeling task.

The study of object orientation in database systems may be pursued by examining three building blocks: structure, behavior, and interaction of objects. Structure and interaction come from database concepts while behavior is specified through programming. Database concepts provide persistent data and programming languages contribute expressive power. Chapter 1 introduces the concept of an object; Chapter 2 is on the structural abstraction of an object; Chapter 3 concerns behavioral abstraction; Chapter 4 discusses object interaction; and Chapter 5 proposes an

object model. For the interested reader, a user guide to the Gemstone object DBMS at Jackson State University is provided in Appendix A. Appendix B contains a program that illustrates the implementation of an object model which has inheritance relationships only. Appendix C is a program that demonstrates the implementation of an object model which has inheritance as well as interclass relationships. Both implementations are on the Gemstone object DBMS. Appendix D gives the outline of a course which presented this tutorial at the Waterways Experiment Station, Vicksburg, MS.

1.2 Object

1.2.1 Structure

An entity is something about which an enterprise stores information. In OO systems, each real-world entity may be modeled by an object. An object is more complex than an entity as the term is used in the entity-relationship model. Like an entity, an object has structure. The structure simply provides descriptive information about the object. For example,

i. an EMPLOYEE object is described by SS#, NAME, SALARY;

ii. a geographic LINE object is described by LINE_ID, START_NODE, END_NODE, DIRECTION, and LENGTH;

iii. a geographic POLYGON is described by POLYGON_ID, POLYGON_TYPE, AREA, and PERIMETER.

iv. an online Word document is described by TITLE, TITLEBAR, SCROLLBAR, BORDERWIDTH, and SIZE.

During modeling, descriptive information that is relevant to the needs of the enterprise is abstracted to specify the structure of an object.

1.2.2 Behavior

In addition to structure, an object may also have one or more algorithms that are applicable to the object. For example, a university enterprise may wish to manipulate

STUDENT objects for calculating a course GPA using an algorithm

$$course\_GPA = (S_A*4.0 + S_B*3.0 + S_C*2.0 + S_D*1.0 + S_F*0.0)/(S_A + S_B + S_C + S_D + S_F)$$

where S_A is the number of students with grade A, and so on.

In traditional environments, the algorithm is written as a program that is separate from the data. In OO systems, the algorithm is part and parcel of the object. Thus the second characteristic of an object is that it possesses behavior that may be invoked for information of interest to the enterprise.

A fundamental and very significant difference to data access between the OO paradigm and traditional data management is given in the following example. Consider the EMPLOYEE table, Figure 1, in a relational database:

EMPLOYEE

SS#	NAME	SALARY	DNAME
777182278	Smith	51000	Applications
179113420	Jackson	62000	DBA

Figure 1. An EMPLOYEE table

The SQL query to retrieve the SALARY of Smith is

```
select SALARY
from EMPLOYEE
where NAME = 'Smith';
```

Note that in traditional data management, the EMPLOYEE relation is separate from the query or application program. Furthermore, the user is required to know the exact names, such as SALARY and NAME, of attributes of the relation. By contrast, if we have an EMPLOYEE object Smith in an OO database, the query would be of the form

Smith *SALARY*

that is, specify the Smith object, then invoke the algorithm *SALARY*. Here, *SALARY* is the name of an algorithm for retrieving the value of salary. The *SALARY* algorithm

is contained in the Smith object. The user does not have to know the attribute names of the object.

In object orientation, application programming is simplified since it often consists of invoking and assembling predefined semantic operations. This is one way in which programmer productivity is improved. Both the value of salary and the algorithm used to retrieve it are embedded in the object itself. This is precisely how the Microsoft GUI, Windows, an OO implementation, operates:

select an object, then select an operation

For example, to delete a Word document text,

select the text, then click the cut command

Compared to the SQL example, the data processing philosophy in OO systems is more natural and in line with other human experiences. For example, when you start a car you identify an object, a key, then you invoke an operation by cranking. To accelerate, you identify the gas pedal and depress it. In each of these the driver does not need to know the underlying mechanism. By contrast, the approach of SQL and traditional data management in general would require knowledge of the internal working of a car. In OO data management, we have persistent data objects to which we may apply named operations.

1.2.3 Interaction

A third characteristic of an object is that it is able to interact with other objects and even with itself. So, for example, a STUDENT object may interact with a COURSE object in that the student takes courses. The relationship, Figure 2, is as much a part of the object as its structure and behavior.

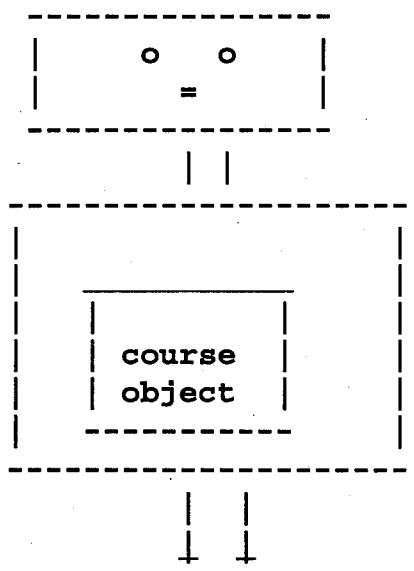


Figure 2. OO STUDENT-COURSE relationship

By contrast, an entity may be involved in relationships which are "external" to it. Figure 3 shows such an occurrence diagram.

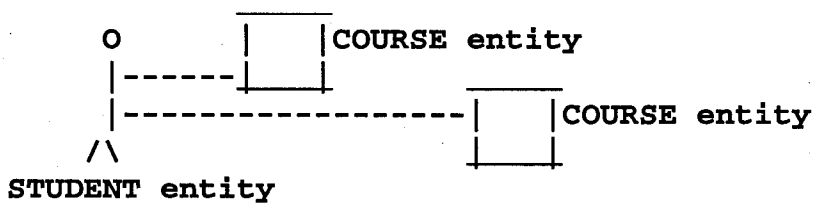


Figure 3. Student with relationships to courses

The corresponding entity-relationship (ER) diagram is given in Figure 4.

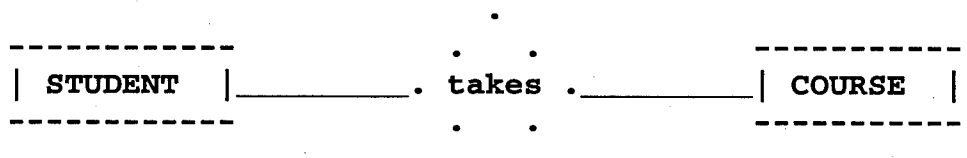


Figure 4. ER STUDENT-COURSE relationship

Unlike in object orientation, the interacting "objects" in the ER and relational models are stored separately, only to

be joined later to respond to queries. Joining exacts a high cost in performance.

1.2.4 Definition

With the preceding background we define an object as:

> a An object is an abstract structural representation of real-world entity that has certain embedded properties for data manipulation and the ability to interact with other objects and itself.

The diagram in Figure 5 illustrates the definition and contrasts it with the more familiar notion of entity.

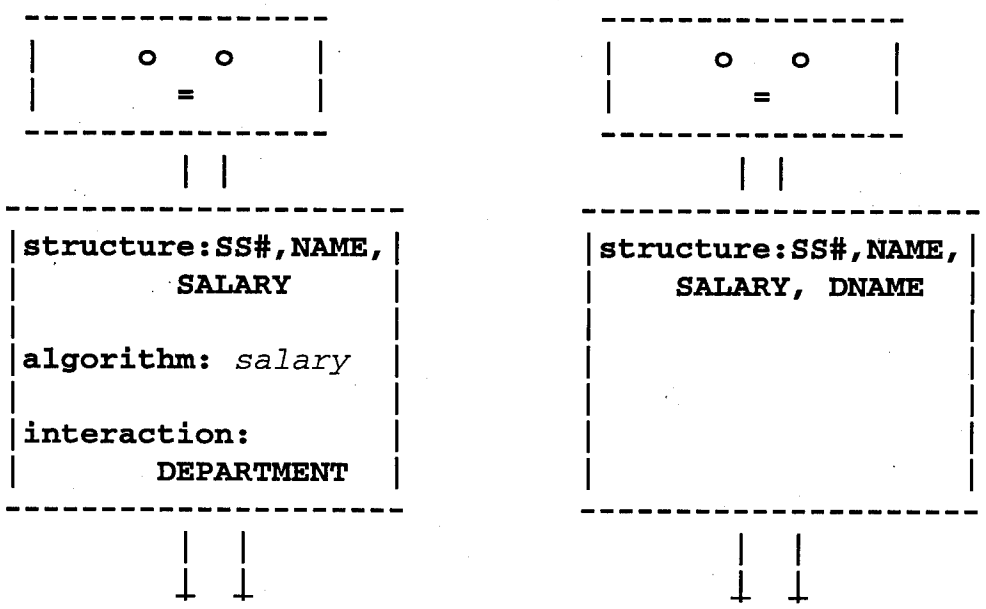


Figure 5. An object and an entity

1.2.5 Object identifier

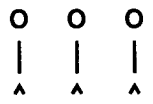
Each object is uniquely identified by an object identifier (OID). An OID is a unique internally generated number which the system assigns to an object the moment it is created and cannot be changed during the lifetime of the object database. The notion of an OID is different from the concept of a primary key in the relational data model. In the latter, a primary key is defined by one or more attributes and uniqueness of tuples in a relation is guaranteed by the value of the primary key. In OO systems, by contrast, two objects are different if they have

different OIDs, even if all their attributes have the same values. The OID can be deleted only if the object is deleted, and the same OID can never be reused in that database. The OID is not tied to a physical address on database disk, allowing for physical data independence in OO systems. The use of OIDs allows objects to share objects and makes possible the construction of general object networks as we shall see in Chapter 4.

Khoshafian and Copeland (1986) describe several techniques for implementing OIDs and conclude that using so-called surrogates is the best technique. Surrogates are system-generated, globally unique identifiers, completely independent of the physical location and data contents of an object.

1.3 Class

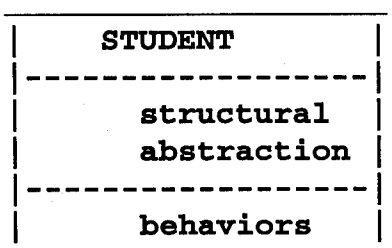
Objects with similar structures, behaviors, and interactions may be grouped into a class. Thus, for example, all the individual STUDENT objects of a university can be abstracted into a STUDENT class, Figure 6.



Individual STUDENT objects:
each has structure, behaviors and
is able to interact with other objects.

Figure 6. STUDENT objects of STUDENT class

We will represent a class with a rectangle, labeled with a descriptive name. For example the STUDENT class is represented as shown in Figure 7.



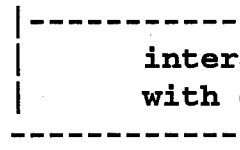


Figure 7. Representation of a STUDENT class

Like an object, a class has the triple:

1. structural abstraction
2. behaviors
3. interaction with objects

An object is the fundamental modeling primitive in object databases. We start by identifying objects which are then grouped, based upon similarity in structure, behavior, as well as interaction, to obtain classes. This is essentially a bottom-up analysis. However, once the classes are defined it is more convenient to take a top-down view of the model during application development. That is, we focus on classes from which individual objects may be created if desired. The class then serves as a template for generating a set of objects that are similar with respect to structure, behavior, and interaction. Thus each object in an OO system is an instance of some class.

An example of the class concept in the Windows environment is as follows. There are many applications, usually represented by icons, of which the Word icon is an example. When you double-click the Word icon, a Word document is created. Similarly many other Word documents may be generated. In this example, the Word application is the class and the Word documents are instances of Word.

1.4 Analogy with Traditional Data Management Terms

To begin to put object concepts into perspective, we list in Table 1 the new terms vis-a-vis familiar terms from other data management approaches.

Conventional data processing	ER model	Relational system	Object-oriented

Field variable	Attribute	Attribute	Instance
Field type	Domain	Domain	Constraint
Record	Entity	Row/Tuple	Object/Instance
File	Entity type	Table/Relation	Class/ADT
_____	Relationship	Foreign key	Relationship
Key	Primary key	Primary key	OID
Procedure	_____	Host lang+DML	Behavior/Method
Procedure call	_____	_____	Message

Table 1. Data management terms

The above analogies are by no means exact, but may help to convey the spirit of the novel concepts.

2. OBJECT STRUCTURE

2.1 Structural Abstraction

The structural abstraction of an object consists of the set of variables that describe that object. For example, a student is described by the variables: SS#, MAJOR, GPA, GRADES, STREET, CITY, STATE, ZIP. These descriptors, shown in Figure 8, are called instance variables.

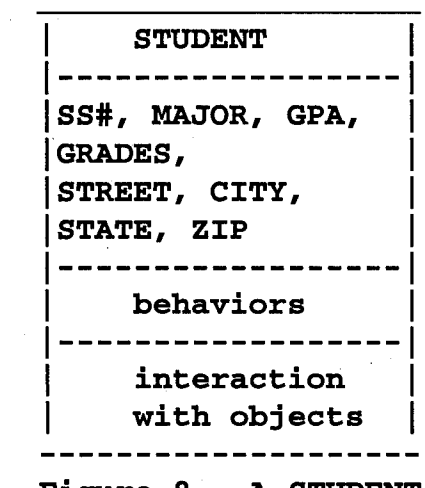


Figure 8. A STUDENT class

Often, instance variables are single-valued for a given object at some point in time. However, an instance variable may be multi-valued as is the case for GRADES. The instance variable GRADES assumes an array of objects as its value. For a particular student, the grades might be A, A, B. A repeating group such as this is not permissible in relational systems but is a hallmark of OO systems. In general, the values assumed by an instance variable may be objects of arbitrary complexity, permitting types such as video, audio, graphics, and digital maps. Thus OO systems are capable of handling complex objects.

2.1.1 Composite instance variable

The user may wish to collect some instance variables into a single unit. This might be justifiable if the instance variables will often be used together. A case in point is the collection of STREET, CITY, STATE, ZIP of our student

example into a single instance variable called ADDRESS. ADDRESS might be intended for purposes such as mailing lists and marketing research. Another situation which might warrant grouping the four instance variables is if they collectively belong to several classes, say, STUDENT, EMPLOYEE, and COMPANY as shown in Figure 9. Under either of these two circumstances, the four instance variables may be defined as a separate new class, ADDRESS, referenced by instance variable ADDRESS of STUDENT, EMPLOYEE, and COMPANY as shown in Figure 10. This is known as referential object sharing. Here, STUDENT, EMPLOYEE, and COMPANY classes share the ADDRESS class.

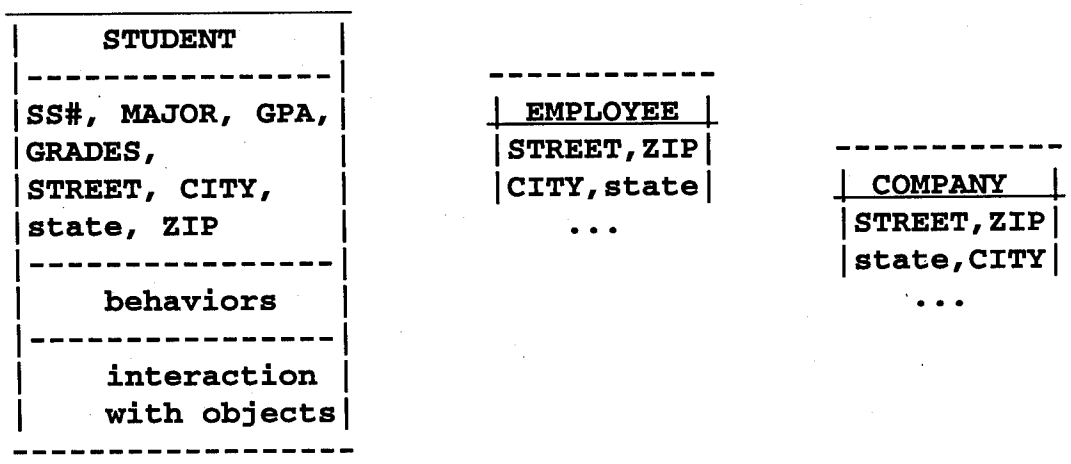
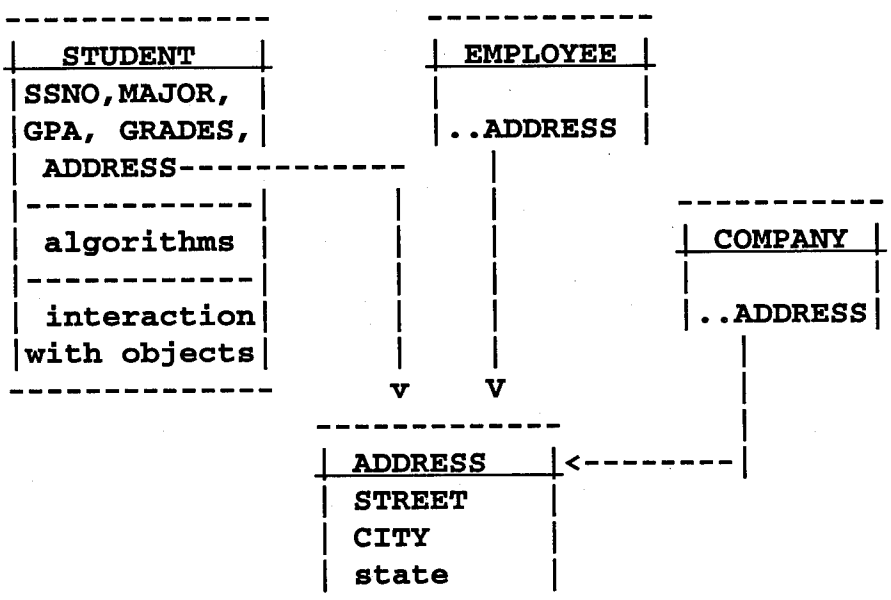


Figure 9. STUDENT, EMPLOYEE, and COMPANY classes



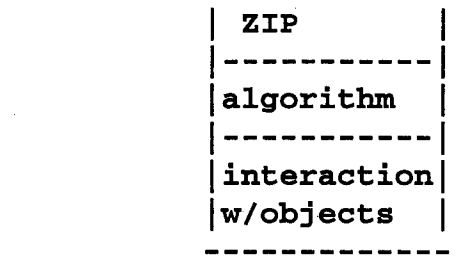


Figure 10. Referential object sharing

2.1.2 Object state

The object state is the set of values that the instance variables assume at a particular time. An example of an object state of the STUDENT class is shown in Figure 11. We shall see that an object state can only be accessed using an algorithm of the given object's class.

STUDENT Class

instVar	object state
SS#	525 37 1995
MAJOR	Computer Science
GPA	2.9
GRADES	A, A, B
STREET	2051 Rancho Alegre
CITY	Pasadena
STATE	CA
ZIP	91109

Figure 11. An object state of STUDENT class

Because of previous usage in relational systems, for example, the terms object state and instance may appear to be synonyms. However, in object orientation, an instance includes the additional notions of behavior and interaction.

2.1.3 Class, instance variables, and object states

To fix our thoughts, it is helpful to start to develop a mental picture of what a class, its instances variables, and states might look like. Accordingly, we use the

familiar table structure to display the class EMPLOYEE, its instance variables, and two of its states.

EMPLOYEE Class

instance Variables	state1	state2 ...
SSNO	E103	E217
ENAME	Smith	Jackson
EXPERTISE	Programmer	DBA

Figure 12. Tabular representation of class

2.1.4 Constraints

Instance variables are constrained to hold certain kinds of objects. Typical constraints are Float, Integer, and String. Constraints provide a way to specify and enforce restrictions on the values of the variable. Constraints may be specified during class creation or modification.

2.1.5 Class definition

To specify a class, the following must be defined:

a) the name of the class
b) each instance variable of the class along with a data type that may be a base data type or an abstract data type
c) the behaviors
d) the interactions

For example, in Gemstone (1994), the structural specification of the EMPLOYEE class may be done as follows.

```
Object suclass: 'EMPLOYEE'
    instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')
    constraints: #[
                 #[#SSNO, String],
                 #[#ENAME, String],
                 #[#EXPERTISE, String]
                  ].
```

We shall study the Gemstone syntax, including the class creation statement, in Chapter 3. Note that behavioral specification is not included in the above class creation statement. Behavioral specification will be covered in Chapter 3.

2.2 Abstract Data Type

Consider the set of integers ...-4, -3, -2, -1, 0, 1, 2, 3, 4, ... These numbers may be viewed as objects. As objects, the numbers have descriptors such as NOTATION(1, 2, 3,...), NAME (one, two, three...), SIGN(+, -), etc. Thus NOTATION, NAME, and SIGN are three of the instance variables of integer numbers. As objects, the integer numbers also have embedded algorithms such as multiplication(*), addition(+), and subtraction(-). Finally, as objects, the integer numbers may interact with other objects. For example,
2 * 3.1 where 3.1 is a real number. Following our earlier analysis, the individual integer objects may be grouped into an INTEGER class, Figure 13.

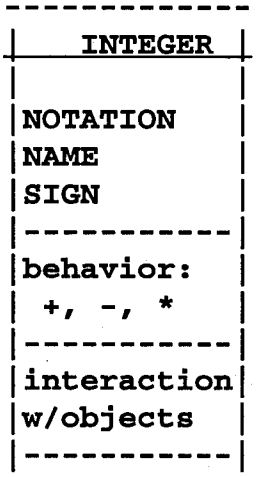


Figure 13. INTEGER class

The INTEGER class has been built into programming languages and DBMSs and therefore the modeler does not normally have to identify and define Integer as a class. Similarly the set of real numbers are objects in the builtin FLOAT class. There is also a STRING class. By convention such builtin classes are called data types.

For the purpose of defining complex objects, the set of conventional data types are rather limited. Thus the approach in object database systems is to augment the builtin data types with user-defined data types. Accordingly, any class defined in an object model may be considered a data type. Like FLOAT, INTEGER, and STRING user-defined classes consist of objects with embedded algorithms and interactions. To distinguish the user-defined data types from builtin data types the former are called Abstract Data Types (ADTs).

In the following example, EMPLOYEE class is an ADT. First EMPLOYEE is defined, then it is used as a data type to constrain the instance variable EMP in the definition of ENROLLMENT.

```
Object suclass: 'EMPLOYEE'
    instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')
    constraints: #[
                 #[#SSNO, String],
                 #[#ENAME, String],
                 #[#EXPERTISE, String]
                  ].

Object suclass: 'ENROLLMENT'
    instVarNames: #('EMP', 'GRADE')
    constraints: #[
                 #[#EMP, EMPLOYEE],
                 #[#GRADE, String]
                  ].
```

2.3 Instance Variable Encapsulation

Instance variables and the internal details of an object are transparent to users and other objects. That is, instance variables are hidden from and cannot be updated directly by users and other objects. They may, however, be updated using the object's own algorithms. This is known as encapsulation.

An example from MS Windows - WINDOWSIZE is an instance variable for the WINDOW class, Figure 14.

However, end users do not know what it is called internally, nor do they know its data type. Yet we can perform an operation such as dragging to alter the value of WINDOWSIZE for a WINDOW instance. Thus the instance variable WINDOWSIZE is encapsulated.

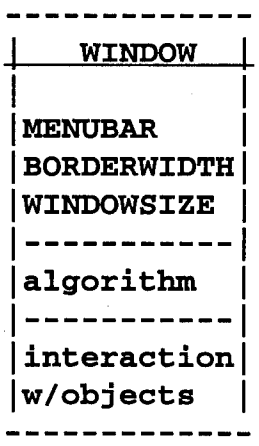


Figure 14. MS WINDOW class

Similarly, when a class is defined in an object database, the instance variables are encapsulated. For example, in the definition of the EMPLOYEE class in the last chapter SSNO, ENAME, and EXPERTISE are encapsulated in a pure object DBMS.
Since the instance variables names are transparent, the only way to access the object state is by means of an algorithm defined for the class. An advantage of this architecture is that the internal representation of the instance variables can be changed without implying changes to the applications that use the class. That is, data independence is provided.

2.4 Class Hierarchy and "IS A"

A class hierarchy is a tree in which each node is a unique class of a database model. Along a hierarchical path, a class at level n+1 is a structural and behavioral subclass of a class at level n. To construct a class hierarchy one identifies a class or several classes that are structural subsets of another class in a model. Such a situation presents an opportunity for introducing a class hierarchy. For example, a class hierarchy may be defined for the classes SECRETARY, SCIENTIST, ENGINEER, and EMPLOYEE. SECRETARY, SCIENTIST, and ENGINEER are all employees and

are therefore subsets of the EMPLOYEE class. The hierarchy is presented as shown in Figure 15.

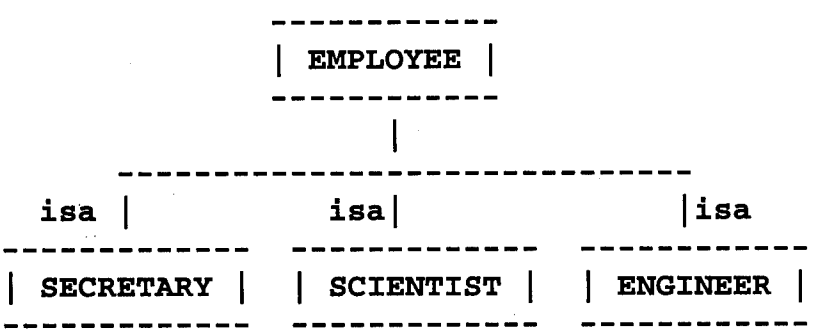


Figure 15. EMPLOYEE class hierarchy

The hierarchical relationship is aptly called "is a". That is,

SECRETARY "is a" EMPLOYEE
SCIENTIST "is a" EMPLOYEE
ENGINEER "is a" EMPLOYEE

At the level of objects, every instance of a subclass is also an instance of the superclass.

A second example of a class hierarchy is:

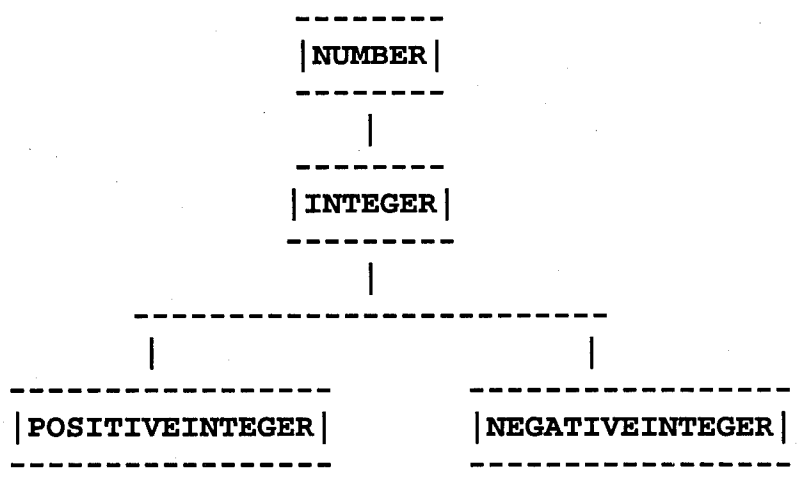


Figure 16. A NUMBER class hierarchy

What structural characteristics make this a class hierarchy?

A third example comes from Windows:

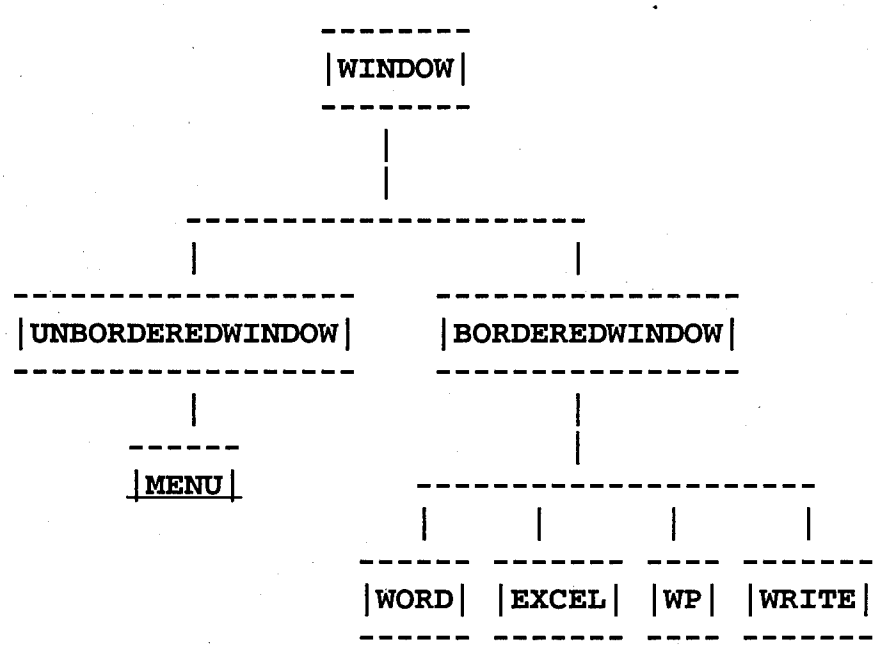


Figure 17. An MS Windows class hierarchy

What structural characteristics make this a class hierarchy?

2.4.1 Generalization or superclass construction

The process of creating a superclass for defined classes is done by abstraction. In Figure 15, we created a superclass, EMPLOYEE, by extracting common general characteristics of certain objects. Such a superclass construction is a process of abstraction that creates a new class. In generalization, the instance variables of the superclass are a subset of the instance variables of any one of its subclasses.

2.4.2 Specialization or subclass construction

Just as a superclass is created by abstraction, subclasses may be created by specializing a class. In specialization, the instance variables of each subclass is a superset of the instance variables of the superclass. SECRETARY, SCIENTIST, ENGINEER are specializations of the class EMPLOYEE, Figure 15.

To create subclasses SECRETARY, SCIENTIST, ENGINEER for EMPLOYEE, the following Gemstone code seems reasonable:

```
EMPLOYEE subclass: 'SECRETARY'
       instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')

EMPLOYEE subclass: 'SCIENTIST'
       instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')

EMPLOYEE subclass: 'ENGINEER'
       instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')
```

However, we shall see in the following section that an object DBMS provides structural inheritance, thereby simplifying subclass definitions.

2.5 Structural Inheritance

The structures for classes SECRETARY, SCIENTIST, ENGINEER, and EMPLOYEE are described using instance variables. Therefore if a SECRETARY is a subset or specialization of EMPLOYEE then the instance variables of SECRETARY and EMPLOYEE must be quite similar. In fact, as stated in the previous section, all the instance variables of EMPLOYEE are applicable to SECRETARY. The same is true for SCIENTIST and ENGINEER classes. In general, in a hierarchy a subclass may inherit some or all the instance variables of the superclass. An important feature of object orientation is inheritance and inheritance of structural properties of a superclass by its subclasses is our first example of this. Suppose the instance variables of EMPLOYEE are as shown in Figure 18.

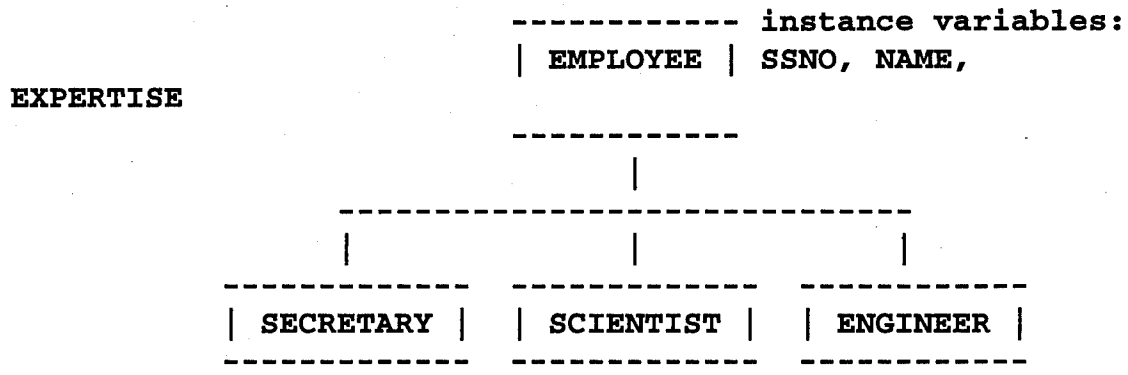


Figure 18. Inheritance in EMPLOYEE class hierarchy

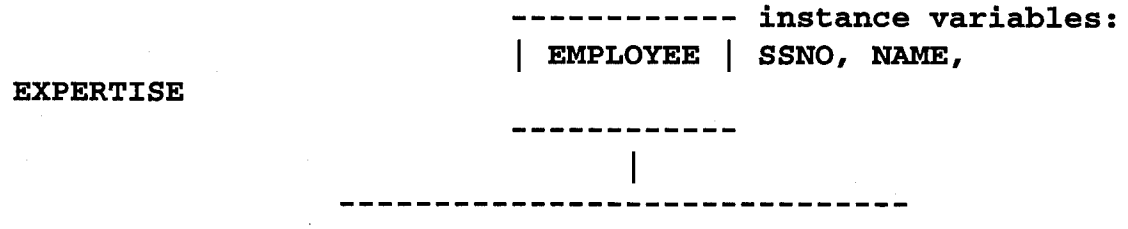
Then SECRETARY, SCIENTIST, ENGINEER would automatically inherit the instance variables SSNO, NAME, and EXPERTISE unless otherwise desired. It would be unnecessary to denote these instance variables for the subclasses in the diagram or even in the code. Thus if EMPLOYEE is defined in Gemstone with the following code:

```
Object subclass: 'EMPLOYEE'
       instVarNames: #('SSNO' 'NAME' 'EXPERTISE')
```

then the code for defining SECRETARY, SCIENTIST, ENGINEER is:

```
EMPLOYEE subclass: 'SECRETARY'
       instVarNames: #()

EMPLOYEE subclass: 'SCIENTIST'
       instVarNames: #()

EMPLOYEE subclass: 'ENGINEER'
       instVarNames: #()
```

In addition SECRETARY, SCIENTIST, ENGINEER may have their proprietary instance variables for specialization as in Figure 19.



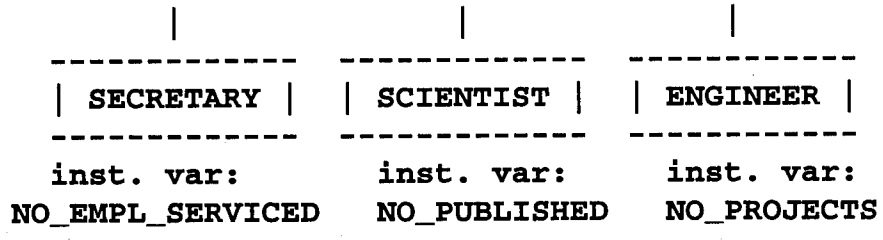


Figure 19. Specialization in EMPLOYEE class hierarchy

The data definition for this model in Gemstone is:

```
Object subclass: 'EMPLOYEE'
       instVarNames: #('SSNO' 'NAME' 'EXPERTISE')


EMPLOYEE subclass: 'SECRETARY'
       instVarNames: #('NO_EMPL_SERVICED')

EMPLOYEE subclass: 'SCIENTIST'
       instVarNames: #('NO_PUBLISHED')

EMPLOYEE subclass: 'ENGINEER'
       instVarNames: #('NO_PROJECTS')
```

The idea of inheritance is so appealing that in pure OO systems, any class you create must be a subclass of another class. In Gemstone, the class OBJECT is the "mother" of all classes. Thus our EMPLOYEE class hierarchy may be redrawn as shown in Figure 20.

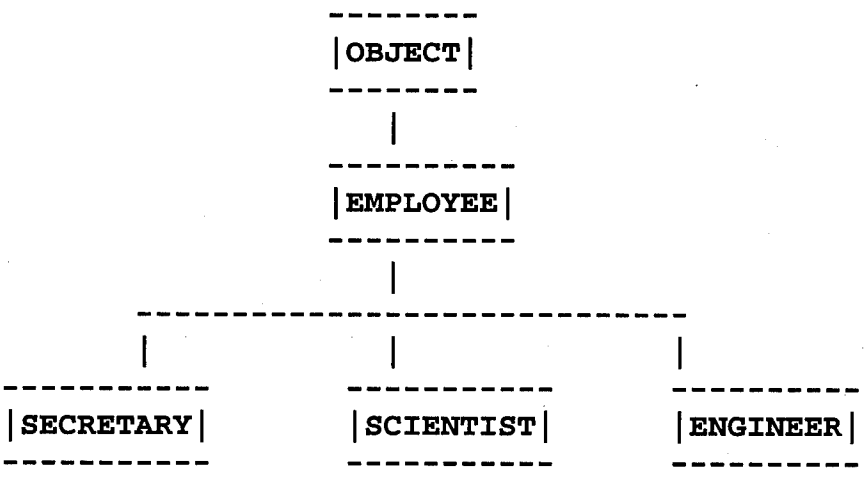


Figure 20. The OBJECT superclass

This explains the mysterious term "Object" in the previous Gemstone data definition for EMPLOYEE:

```
Object subclass: 'EMPLOYEE'
       instVarNames: #('SSNO' 'ENAME' 'EXPERTISE')
```

EMPLOYEE inherits from OBJECT a storage format that enables it to define named instance variables, etc. In turn, EMPLOYEE can also define specialized variables that can be

exploited by its subclasses. For example, it would endow its instances with storage slots called SSNO, ENAME, and EXPERTISE. Thus an important part of designing an OO class is choosing its superclass. If someone else has defined a class that provides some of the properties you need, you can save a lot of time by using it as a superclass. Thus, your class inherits proven data structures that you can tailor to fit your own needs.

An example of a class hierarchy in the Windows architecture is shown in Figure 21.

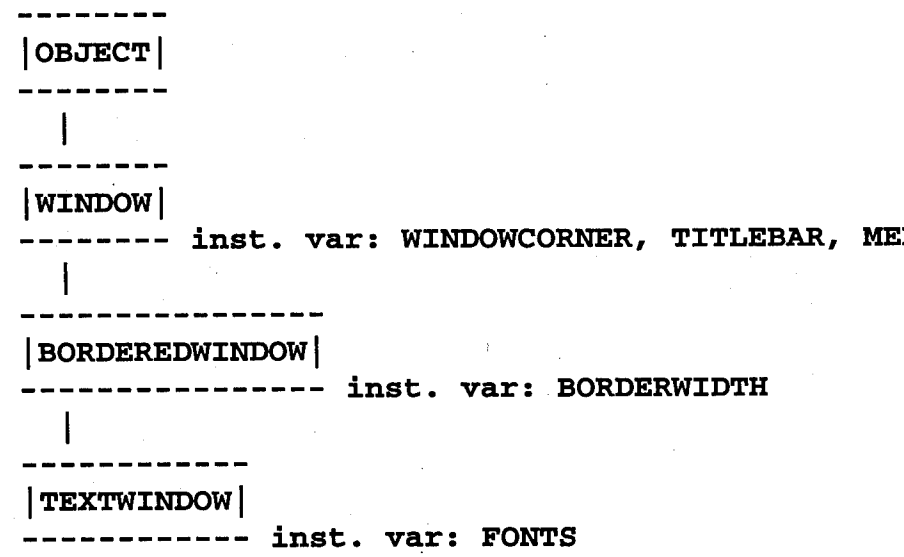


Figure 21. Inheritance in a Windows class hierarchy

In this example, BORDEREDWINDOW and TEXTWINDOW inherit WINDOWCORNER, TITLEBAR, MENUBAR, ... from the WINDOW class.

2.6 Instance Variable Overriding

To be able to monitor the values entered into instance variables during initialization or update, constraints are defined for each instance variable. Thus the variable is restricted to contain only the specified kind of object. Constraints are like domains in relation systems and define the set of allowable values an instance variable may assume. Like instance variables, constraints may be inherited by subclasses. In the subclass, a constraint on an inherited instance variable may be changed, thus enabling an override of the superclass instance variable.

However, note that the inherited instance variable's constraint must be a subclass of the inherited constraint. For example, SmallInteger is a subclass of Integer and, therefore, SmallInteger may be used in a constraint to override Integer as in the following data definition of the class hierarchy of Figure 22.

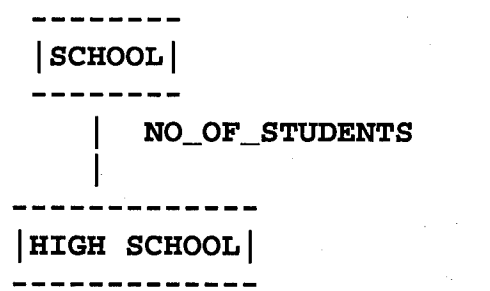


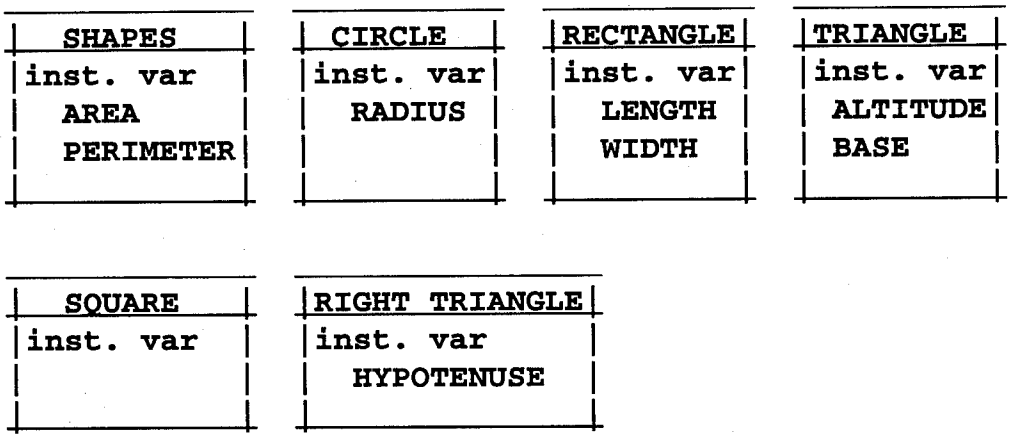
Figure 22. School/High-School class hierarchy

The Gemstone data definition is

```
Object subclass: 'SCHOOL'
       instVarNames: #('NO_OF_STUDENTS' ...)
       constraints:  #[
                     #[#NO_OF_STUDENTS, Integer],
                      ...
                      ].

SCHOOL subclass: 'HIGH_SCHOOL'
       instVarNames: #('NO_OF_STUDENTS' ...)
       constraints:  #[
                     #[#NO_OF_STUDENTS, SmallInteger],
                      ...
                      ].
```

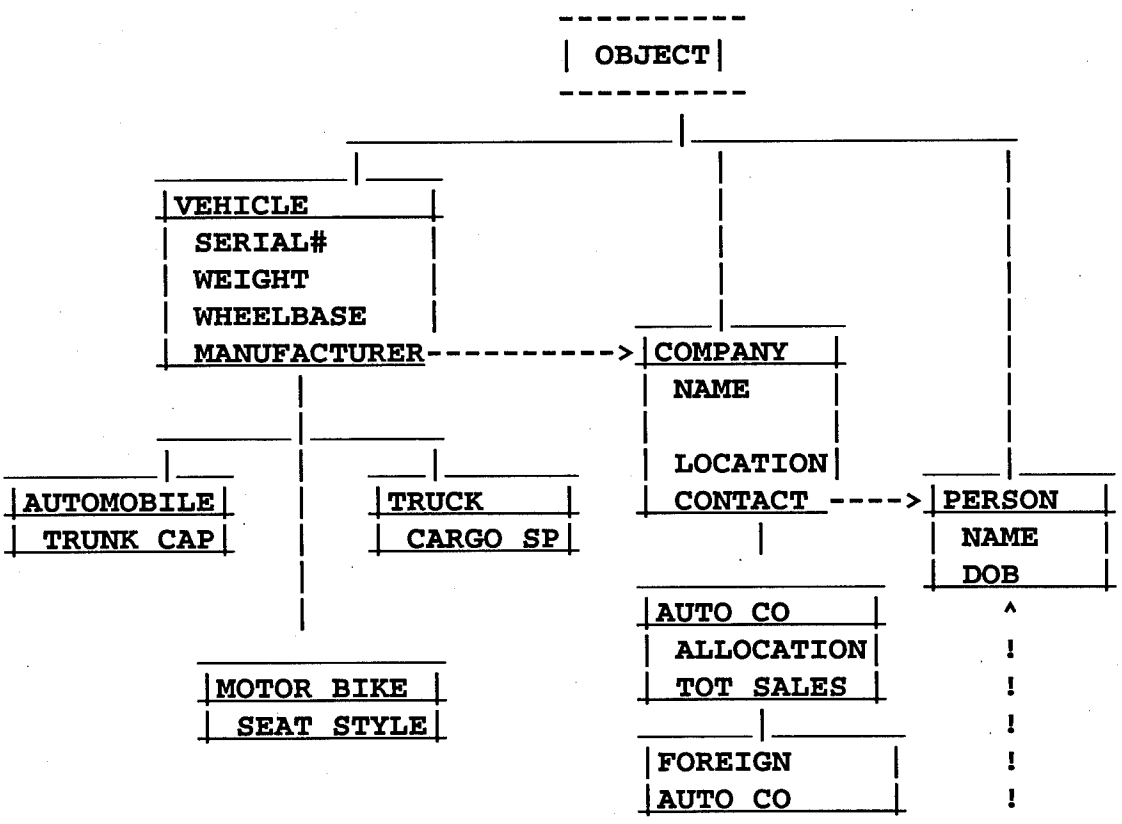
Assignments:

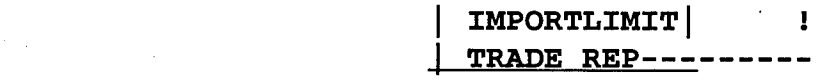
1. You wish to build an application that uses graphic objects and you have specified the following classes:



a) Draw the class hierarchy for the graphic objects.
b) List all the instance variables of the RightTriangle class.

2. Consider the object model for a VEHICLE database (McFadden and Hoffer 1991):





_____ inheritance relationship
-------> aggregation relationship for composite instance variables

Write a Gemstone data definition for VEHICLE, AUTOMOBILE, and TRUCK only.

3. Specify a class and define two subclasses for it to illustrate instance variable inheritance, overriding, and specialization. Give the Gemstone data definition.

3. OBJECT METHOD

3.1 Behavior

We have seen in the previous chapter that instance variables carry structural and, as we shall see later, interrelationship information about the objects of a class. Thus, in order to manipulate an object, one must further process the values of one or more of its instance variables. This processing is done through algorithms to express the behavior of an object. In traditional applications, algorithms that operate on data values of objects to provide information are packaged as procedures and form part of programs that manipulate the values. In OO methodology these algorithms, called methods, are an integral part of the objects themselves. Methods manifest object behavior. For example, the EMPLOYEE class in Figure 23 has the instance variables SSNO, NAME, DOB, and the method *current_age*.

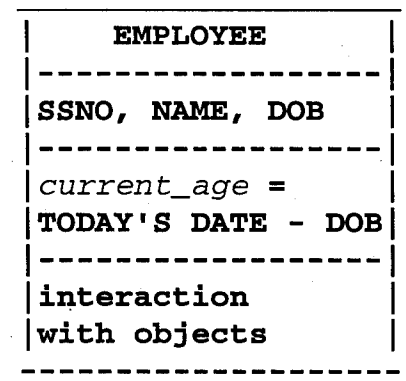


Figure 23. EMPLOYEE class

The method *current_age* computes the current age of EMPLOYEE objects using values of the variables TODAY'S DATE and DOB. As in this example, all method names will be italicized.

The following are two examples, from Microsoft Windows, of methods in action:

a) When you start the Word application and click File New, a new instance of the WORD class is created. This new instance may be used to prepare a Word document. Here, *new* is a method of the WORD class. The method *new* is as much a part of Word as the Borders and Icon that

structurally represent Word. Also note that you have no access to the code that implements the method *new*.

b) When you select a paragraph of a Word document and click Edit Copy, a method is invoked that copies the selected object to the clipboard. Again the method *Copy* is a part and parcel of Word. Also note that you have no access to the code for the method *Copy*.

By contrast, relational tables do not have methods associated with them.

3.2 Kinds of Methods

There are two kinds of methods: instance methods and class methods.

3.2.1 Instance method

Instance methods are methods that enable instances to respond to messages. In the specification of their algorithms, instance methods usually refer to the instance variables of a class. For example, in Section 3.1 the method

current_age = TODAY'S DATE - DOB

Here *current_age* is an instance method that refers to instance variable DOB in its algorithm.

Instance methods are applicable to and are understood by instances.

3.2.2 Class method

A class method is one that is understood by a class, but not by instances. For example, most classes contain the method *new* which is used to create instances for the class. Class methods are applicable to and are understood by classes.

3.3 Inheritance of Methods

The statements already made regarding subclass/superclass construction for inheritance of instance variables apply equally well to the inheritance of methods. A subclass inherits both the instance variables and methods of its superclass. At times, instance variables drive the construction of the hierarchy. For example, structuring SECRETARY as a subclass of EMPLOYEE is a design that is driven by a consideration of instance variables. Consider the class hierarchy in Figure 24.

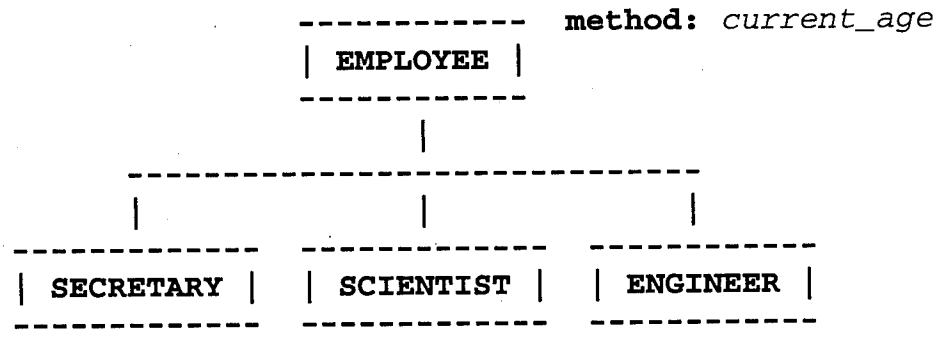


Figure 24. EMPLOYEE class hierarchy and method inheritance

Even though the hierarchical construction may have been driven by instance variables, advantage is taken of the structure to inherit methods. In the example, each of the subclasses SECRETARY, SCIENTIST, and ENGINEER inherits the method *current_age*. *Current_age* is first written for EMPLOYEE and due to inheritance, it may be reused by SECRETARY, SCIENTIST, and ENGINEER. Hence, the concept of inheritance provides a reusability mechanism.

At other times, it is methods that motivate the structuring of class hierarchies. For example, structuring INTEGER as a subclass of NUMBER in Figure 25 is based upon a consideration of the methods that INTEGER would inherit from NUMBER.

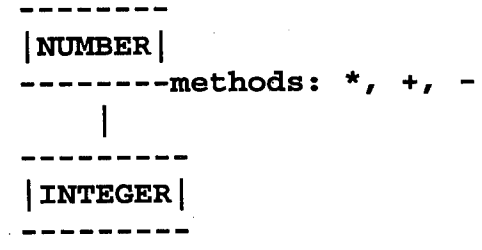


Figure 25. A NUMBER class hierarchy

This latter example shows that in addition to user-defined methods such as *current_age*, certain builtin methods are provided. Thus important and generally applicable methods such as scalar comparisons, arithmetic operations, and string manipulations are part of the system and may be inherited along a hierarchical path. For example, multiplication(*) and addition(+) which may initially have been defined for the NUMBER superclass may be inherited by the INTEGER subclass.

An important builtin method which was discussed previously is the method *new*. *New* is a method of the root class, OBJECT. Applying *new* to a class creates a new object in the class. Through inheritance a builtin method such as *new* is available to every other class in the hierarchy. For example, in Windows the subclasses WORD, EXCEL, WP, WRITE inherit the method *new* from OBJECT, Figure 26. Thus each of these applications may use the method *new* to create new instances, that is, new documents of the various applications.

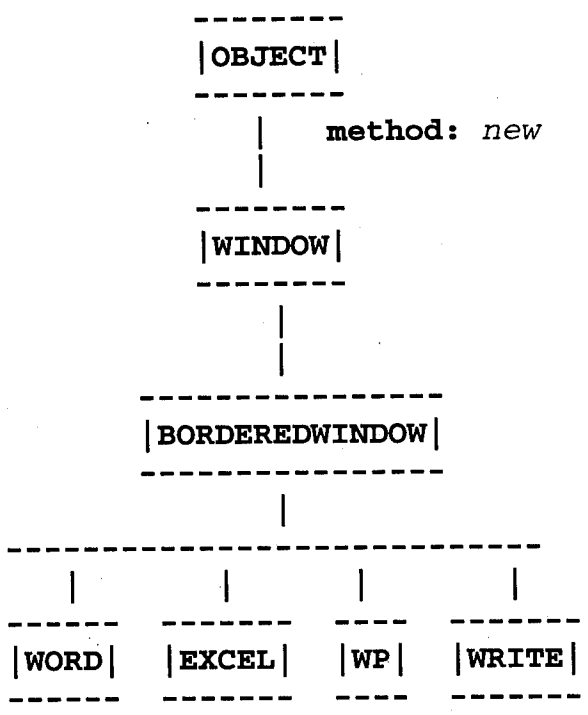


Figure 26. A Windows class hierarchy

Consider also method inheritance in the Windows class hierarchy in Figure 27. *MOVEWINDOW, RESIZEWINDOW* **are inherited by and are applicable to the subclasses BORDEREDWINDOW and TEXTWINDOW.**

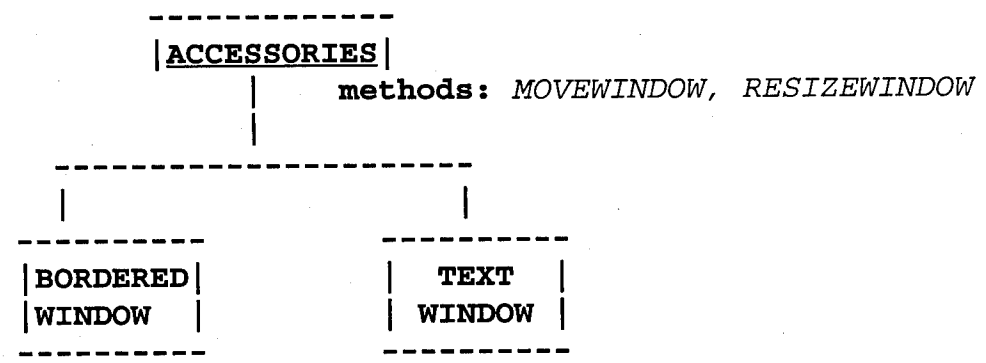


Figure 27. Windows ACCESSORIES class hierarchy

3.4 Method Inheritance Algorithm

When the system invokes a method of an object, the entire hierarchical path of the object is searched for the matching method, using the following sequence:

1. Scan the class to which the object belongs
2. If the method is not found, scan the superclass
3. Repeat Step 2 until the method is found, or
4. The top of the class hierarchy is reached without finding
   the message. In this case the system will generate a message to indicate that the method was not found.

To illustrate the scanning process, let's examine the EMPLOYEE class hierarchy in Figure 28.

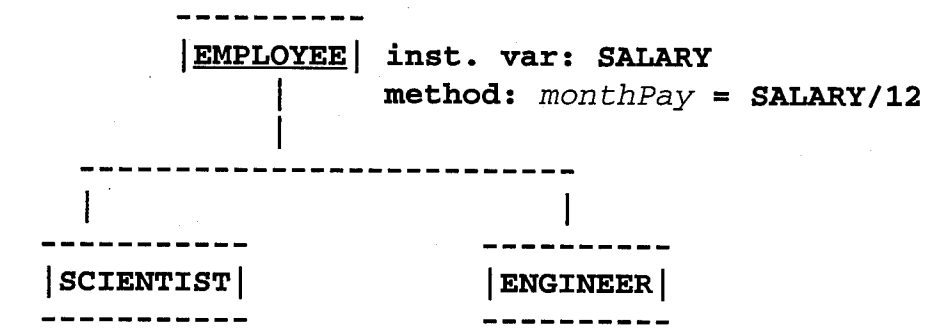


Figure 28. Method lookup in EMPLOYEE class hierarchy

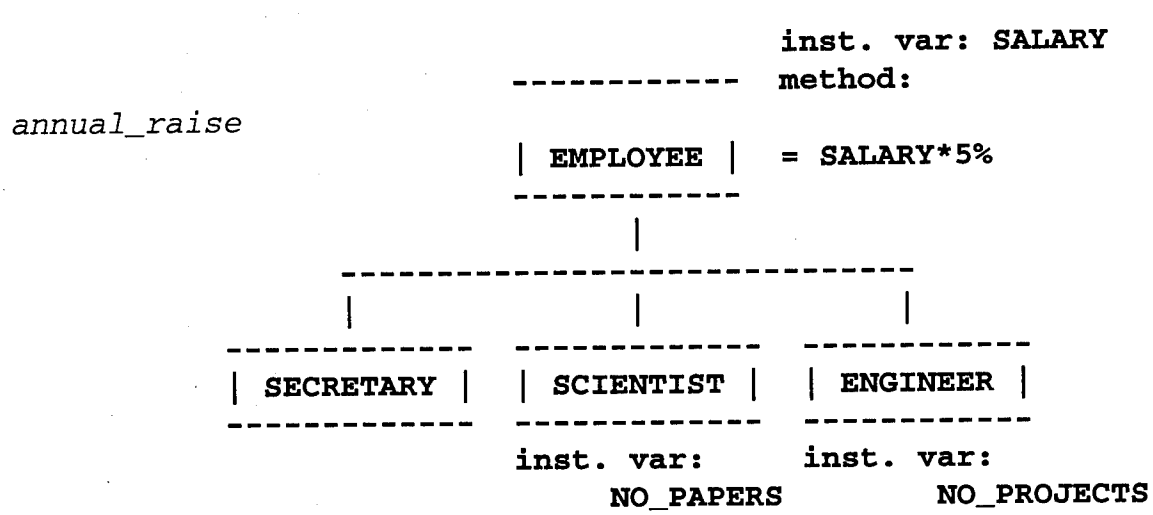
If we invoke the *monthPay* method of a SCIENTIST instance, it will execute the *monthPay* method defined in its EMPLOYEE superclass. This happens despite the fact that *monthPay* is not explicitly defined as a method for SCIENTIST class.

Note the code reusability benefits obtained through OO systems: the *monthPay* method's code is available to both SCIENTIST and ENGINEER subclasses and to their subclasses.

3.5 Method Overriding

The inheritance procedure given in the last two sections appears to suggest that it is mandatory for a subclass to inherit the method of a superclass. Yet, it is sometimes desirable for the subclass to override the definition of the superclass methods. Accordingly, the inheritance mechanism allows a class to specialize superclass methods by additions and substitutions. If the name of a method explicitly defined in a class is the same as a method of a superclass, the method from the superclass is not inherited; that is, the definition in the subclass which

likely gives a different behavior overrides the superclass definition. Consider the salary raise example in Figure 29.



(SCIENTIST) method: *annual_raise* **= super** *annual_raise* **+ SALARY*NO_PAPERS/100**

(ENGINEER) method: *annual_raise* **= super** *annual_raise* **+ SALARY*NO_PROJECTS/100**

Figure 29. Overriding in EMPLOYEE class hierarchy

In the salary raise example, when the method, *annual_raise*, **is invoked for SECRETARY the value returned is computed from**

annual_raise **= SALARY*5%.**

This is because SECRETARY inherits the method of superclass EMPLOYEE. However, when the method, *annual_raise*, **is invoked for SCIENTIST the value returned is computed from**

annual_raise **= super** *annual_raise* **+ SALARY*NO_PAPERS/100**

This is because the annual raise method defined for SCIENTIST overrides that of the EMPLOYEE superclass.

Similarly, the method, *annual_raise*, **invoked for ENGINEER returns the value computed from**

annual_raise **= super** *annual_raise* **+ SALARY*NO_PROJECTS/100**

Note that the pseudovariable super represents the superclass. The use of super changes the method lookup as follows:

1. **Scan the superclass**
2. **Repeat Step 1 until the method is found, or**
3. **The top of the class hierarchy is reached without finding the message. In this case the system will generate a message to indicate that the method was not found.**

Figure 30 is an example from Microsoft Windows in which the method *Copy* **of the superclass Program Manager makes copies of application icons whereas the method** *Copy* **of subclass WORD makes copies of text to the clipboard.**

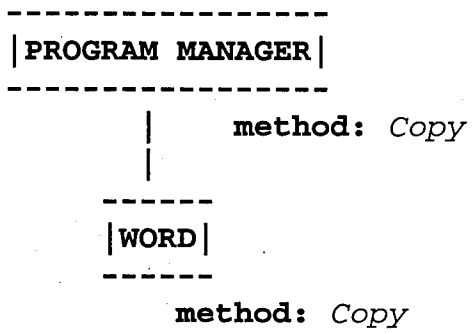


Figure 30. Overriding in MS Windows

3.6 Polymorphism and Dynamic Binding

In the last section, we saw that the same method name may be used by different classes and that those methods may operate differently depending upon the class involved. This phenomenon is known as polymorphism.

Polymorphism is achieved through dynamic (or late) binding. Dynamic binding means that the system will bind a method name at runtime to the appropriate method. Note that it is not possible to do this neatly in the absence of dynamic binding using procedure names in conventional programming.

In the latter environment, the desired effect can only be achieved using case statements. The example in Appendix B illustrates polymorphism.

3.7 Message

In traditional programming languages, procedures are invoked by "calls". Correspondingly, OO methods are invoked by messages. To activate a method of an object, a message is sent to that object. On receipt of the message, the method is executed and a result is returned to the sender. Activating an object's method involves

1. Specifying the target object
2. Specifying the name of the method
3. Specifying the arguments of the method, if any
4. Internally, calling the code that implements the method and binding the parameters to the actual arguments of the invocation

For a given method, a message pattern exists for specifying 1, 2, 3. Message patterns are discussed next in the "Introduction to Gemstone".

**********Introduction to Gemstone **********

The Gemstone Message Syntax

Message Expressions - A message expression consists of:

- **an identifier or expression representing the object to receive the message. The latter is called the receiver.**
- **one or more identifiers called selectors that specify the message to be sent. A selector is a method name.**
- **zero, one, or more arguments that pass information with the message**

The general syntax is

receiver selector [argument]

(That is, object *methodname* **[argument])**

Messages are of three kinds, classified according to the kinds and the number of their selectors and arguments.

When an object receives a message, it compares the incoming message with all those that its class defines and invokes the one whose message pattern matches the arriving message.

Unary Messages

Consists of a receiver and a single selector.

EMPLOYEE *new* **"Class EMPLOYEE is the receiver and** *new* **is the unary selector. This message expression creates an instance of EMPLOYEE"**

An example from Windows - Create a new document with

Double-click Word, click New

This is equivalent to the message

WORD *new*

Binary Messages

Consists of a receiver, a single selector, and a single argument.

2 + 8

The object 2 from INTEGER class is the receiver, + is a binary selector, and 8 is the argument. When 2 sees the selector +, it looks up the selector in its private memory or protocol and finds the method to add the argument 8 to itself.

An example from Windows - While in a Word document, choose Paste. Text residing on the Clipboard, if any, is inserted into the current text. This could be paraphrased

WorkingDocument *Paste* **ClipboardText**

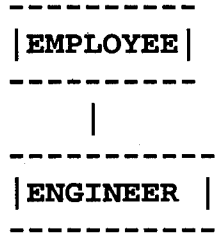
Here WorkingDocument is the receiver, *Paste* **is a binary selector, and ClipboardText is the argument.**

Keyword Messages

A keyword is a simple identifier ending in a colon.

EMPLOYEE *subclass:* **'ENGINEER'**

Here EMPLOYEE is the receiver, *subclass:* **is a keyword selector, and 'ENGINEER' is the argument. The method invoked by the message constructs class hierarchies. In our case, the hierarchy**



is constructed.

An example from Windows - Renaming a file. First you highlight the receiver object(a file). Then you click the

keyword selector *Rename:*, and then you enter the NewFileName ,an argument, in the dialog box. This is equivalent to the message

```
FileName Rename: 'NewFileName'
```

Temporary Variable - Gemstone requires you to declare new variable names before using them. To declare a temporary variable, you surround it with vertical bars as in

```
|myTemporaryVariable|

myTemporaryVariable := 2.
```

Usage: to manipulate an object you need to assign the object (actually the object's OID) to a temporary variable. For example,

```
|Alex|

Alex := EMPLOYEE new
```

Returning Values - whenever a message is sent, the receiver of the message returns an object. This object is the value of the message expression. An assignment statement can be used to capture a returned object:

```
     |myVariable|
     myVariable := 8 + 9.    "the binary message 8 + 9
                               returns object 17. Next,
                               assign returned object 17 to
                               myVariable"
     myVariable              "return the value of
myVariable"

       17                    "returned value"
```

Structure and Examples of Methods

The first and last lines delimit the method:

```
method: EMPLOYEE                    "first line"
message pattern

     (code)

%                                   "last line"
```

The first line indicates that this code is a method and it is for the EMPLOYEE class. The last line (%) is a terminator. "Message pattern" gives the form in which the method will be invoked. "Code" gives an implementation of a behavior of the object.

To invoke this method apply the message

```
     (EMPLOYEE instance) message pattern
```

A. A Simple Method for Testing

The following example defines the simplest possible method for the EMPLOYEE class:

```
method: EMPLOYEE
isEMPLOYEE                    "the message pattern"

     ^true                         "a return statement"
%
```

The message pattern in the method is *isEMPLOYEE*. A statement preceded by a caret(^) returns its value to the expression that invoked the method.

When an instance of EMPLOYEE receives the message *isEMPLOYEE*, it activates the method defined above, which then returns the value true. For example,

```
| Alex |

Alex := EMPLOYEE new.       "create an instance of EMPLOYEE
                             using a unary message and assign
                             the new instance to the variable
                             Alex"

Alex isEMPLOYEE                  "a unary message"

true                          "returned value"
```

B. Methods that use Arguments

Method arguments correspond roughly to subprogram arguments or parameters. They enable you to pass information (objects) to a method, just as subprogram arguments let you pass information to a subprogram. The following example defines for EMPLOYEE a method requiring a single argument:

```
method: EMPLOYEE
areYou: anObject              "the message pattern"

                              "If the argument is the string
                               'EMPLOYEE', return true.
                               Otherwise return false."

     ^anObject = 'EMPLOYEE'
%
```

In the example, the token anObject is like a formal parameter or "dummy variable." At execution, Gemstone replaces it with an actual value. In the following invocation of *areYou:*, anObject takes on the value 'Airplane'

```
| Alex|

Alex := EMPLOYEE new.

Alex areYou: 'Airplane'          "a keyword message"

false                              "returned value"
```

In the following invocation of *areYou:*, anObject takes on the value 'EMPLOYEE'

```
| Alex|

Alex := EMPLOYEE new.

Alex areYou: 'EMPLOYEE'

true                               "returned value"
```

C. Methods for Initializing/Updating and Retrieving

Recall that the instance variables of EMPLOYEE are SSNO, ENAME, EXPERTISE.

In the following, we write the method *SSNO: for initializing instance variable SSNO.*

```
method: EMPLOYEE
SSNO: aNumber                "the message pattern"

     SSNO:= aNumber          "assignment to inst. var: SSNO "
%
```

Note that the method *SSNO:* is reusable by all instances of EMPLOYEE and by instances of subclasses of EMPLOYEE.

In the following, we write the method *SSNO* for retrieving values of instance variable SSNO.

```
method: EMPLOYEE
SSNO            "the message pattern"

     ^SSNO      "return the value of the inst. var 'SSNO' "
%
```

Note that the method *SSNO* is reusable by all instances of EMPLOYEE and by instances of subclasses of EMPLOYEE.

The following code uses methods *SSNO:* and *SSNO respectively* to change the value of instance variable 'SSNO' and to retrieve the new value.

```
| Alex |

Alex := EMPLOYEE new

Alex SSNO: '555239946'    "a keyword message which assigns
                           the number 555239946 to new
                             EMPLOYEE Alex"
Alex SSNO                    "a unary message to retrieve
the
                             SSNO of the new EMPLOYEE"

555239946                    "returned value"
```

Similar to the method for *SSNO:* and *SSNO*, the following two sets of methods need to be written for EMPLOYEE. This is left as an exercise.

ENAME:/EXPERTISE: and ENAME/EXPERTISE

Respectively, these two sets of methods may be used to initialize and retrieve values of the corresponding instance variables.

The following three messages may be used to initialize Paul instance of EMPLOYEE -

```
| Paul |

Paul := EMPLOYEE new

Paul SSNO: '213415678'.
Paul ENAME: 'Paul'.
Paul EXPERTISE: 'Teller'.
```

This may be more conveniently done through cascading.

Cascaded Messages - to send a series of messages to the same object, cascade the messages as in

```
Paul SSNO: '213415678'; ENAME: 'Paul'; EXPERTISE: 'Teller'.
```

In either case, the object state is

Instance variable	State

SSNO	213415678
ENAME	Paul
EXPERTISE	Teller

Note:
Appendix A contains a user guide to the Jackson State University Gemstone ODBMS.
**********End Gemstone Syntax Preview**********

3.8 A Perception of Instance Variables, Instances and Methods

To fix our thoughts, we continue to develop the picture, first given in Section 2.1.3, of what a class, its instances variables, instances, and methods might look like. We use a table structure to display the EMPLOYEE class, its instance variables, some of its instances, a class method, and instance methods. In Figure 31, the receiver of the class method *new* is the EMPLOYEE class. The instance methods are applicable to the various EMPLOYEE instances.

When an object receives a message, it compares the incoming message with all those that its class defines and invokes the one whose message pattern matches the arriving message.

EMPLOYEE Class classmethod *new, ...*

instVar	**instance1**	**instance2**
inst3....		
SSNO	**E103**	**E217**
ENAME	**Smith**	**Jackson**
EXPERTISE	**Programmer**	**DBA**

inst. methods	
areYou	---->
isEMPLOYEE	---->
SSNO:	**---->**
SSNO	---->
ENAME:	---->
ENAME	---->
EXPERTISE:	---->
EXPERTISE	---->

Figure 31. Tabular structure of EMPLOYEE class

As stated previously, instance and class methods are part and parcel of the class in much the same way as instance variables are. When an object receives a message, it compares the incoming message with all those that its class defines and invokes the one whose message pattern matches

the arriving message. The next section elaborates on the nature of methods of a class.

3.9 Protocol

A given class may have many messages, each message corresponds to a single method. The collection of messages, each identified by a message pattern and accompanied by a detailed explanation of its usage, make up the class protocol. For example, we have in Table 2 three categories of methods for the EMPLOYEE class

EMPLOYEE Initialization/Updating Protocol

Message pattern	Usage
SSNO: **aString**	**Initializes inst. var. SSNO to aString.**
ENAME: **aString**	**Initializes ENAME to aString.**
EXPERTISE: **aString**	**Initializes EXPERTISE to aString.**

EMPLOYEE Retrieval Protocol

Message pattern	Usage
SSNO	**Retrieves value of inst. var. SSNO.**
ENAME	**Retrieves value of ENAME.**
EXPERTISE	**Retrieves value of EXPERTISE.**

EMPLOYEE Testing Protocol

Message pattern	Usage
isEMPLOYEE	Test if a given object is an EMPLOYEE. Returns true if object is an EMPLOYEE.
areYou: anObject	Speculate about what an object is. Returns true if the argument is the string 'EMPLOYEE'. Otherwise return false.

Table 2. An example protocol

User-defined methods may be put in one or more categories. The protocol is also known as the object's public aspect. Other objects and users know the class by its protocol. Given the protocol, the user knows the function of a method, the message pattern with which to invoke the method, but does not have access to the code of the method. When an object receives a message, it compares the incoming message with all those that are in its protocol and invokes the one whose message pattern matches the arriving message.

3.10 Encapsulation

The internal representation of instance variables and methods of the class is hidden from other objects and users. The implementation of instance variables and methods constitutes the object's private aspect. The hiding of information is achieved through presenting the user with protocols that specify and define available message patterns. In pure OO systems, a user's interaction with data is confined to only messages defined from the public aspect. Limited to this mode of database access, the instance variables and the methods' code need not be made available to the user. Thus encapsulation is achieved for methods and instance variables.

Encapsulation makes the internal structure (the instance variables, their data values, and the methods) of the class transparent to the user and other objects. As such, the only way to access the object state is by means of messages defined using the class protocol. For example, for the user-defined STUDENT class, we may define the methods *add_course*, *avg_GPA*, etc. as integral parts of the STUDENT class. Subsequently, these methods may be executed on demand by appropriate messages. An advantage of this is that the internal representation such as the instance variables can be changed without implying changes to the applications that use the class. That is, data independence is provided. Thus, in OO systems data independence is achieved through the structuring of public and private aspects for objects (Cattell 1994). Contrast this with the three-level ANSI/SPARC architecture that provides data independence in conventional database systems.

Assignment:

1. Study the employee object database in Appendix B for the implementation of the concepts discussed so far.

2.

a) Specify a class and define subclasses to illustrate method inheritance and overriding.

b) Give a data definition for the superclass in a).

c) Write methods to initialize the instance variables of the class in b).

d) Using a temporary variable, create a new instance for the class in c) and initialize its instance variables.

3. Write a method to create a new employee object in the employee database in Appendix B and return the new object.

4. Which is easier for the modeler, structural or behavioral abstraction? Discuss.

4. OBJECT INTERACTIONS

The previous two sections addressed the structures and behaviors of classes. This section discusses interaction of a class with other classes. Interactions are important in database systems. In fact, the implementation and subsequent navigation of relationships of objects is the most compelling reason for developing database systems. Of the various kinds of database systems, object technology offers and exploits the most comprehensive set of interactions (Loomis 1995). There are three general kinds of interactions in object databases (Rob and Coronel 1993):

i. Inheritance (or hierarchical) Relationship
ii. Interclass Relationship
iii. Aggregation Relationship

The three kinds of interaction are discussed below.

4.1 Inheritance Relationship

The primary interaction in an object model is the class hierarchy. The class hierarchy concept as well as its major function of inheritance have been fully discussed in Chapters 2 and 3. Basically, the construct provides code reusability, an important justification for object orientation. The construction of a hierarchy follows the steps:

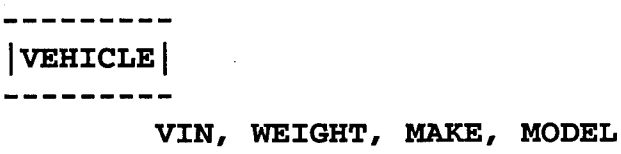
- choose a superclass from among the existing classes
- recursively specify subclasses from the remaining subclasses
- if appropriate, create more specialized subclasses for the original classes

For example, suppose we start with the classes

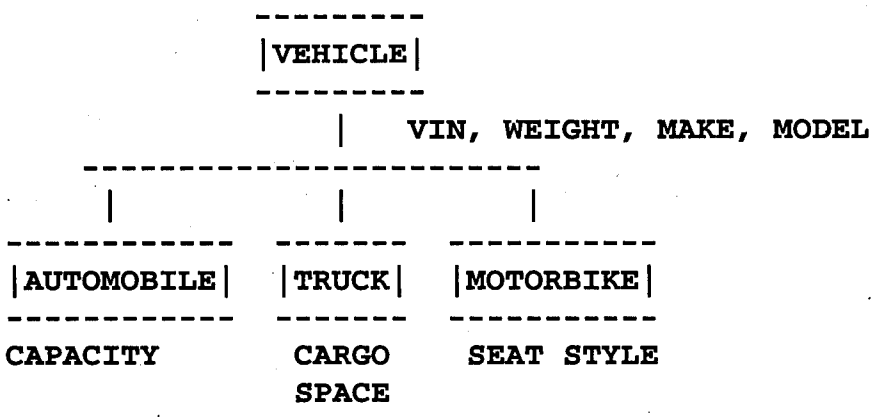
TRUCK, MOTORBIKE, VEHICLE, and AUTOMOBILE

Then

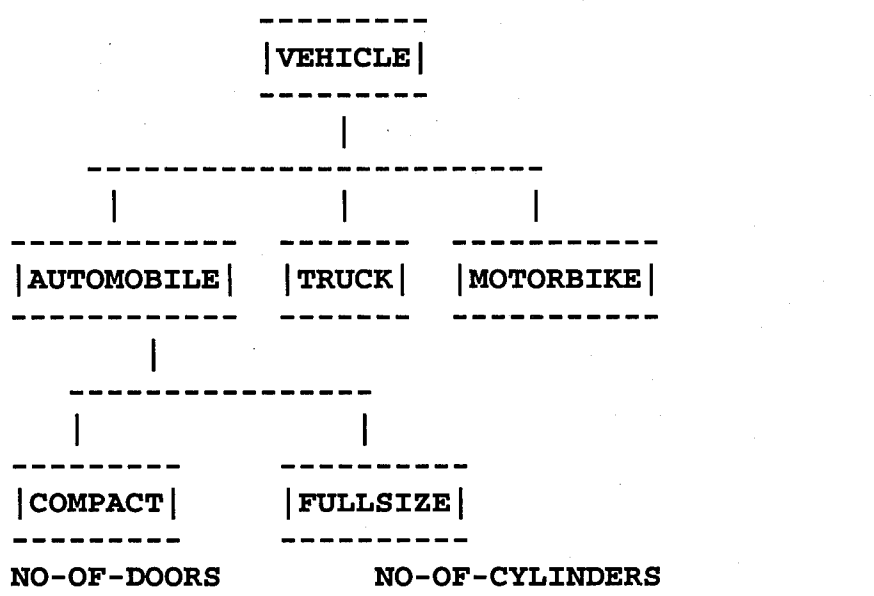
Step1: Based upon the fact that each of AUTOMOBILE, TRUCK, MOTORBIKE "is a" VEHICLE, we select VEHICLE as the superclass



Step2: Make AUTOMOBILE, TRUCK, and MOTORBIKE subclasses of VEHICLE



Step3: At this point we introduce COMPACT and FULLSIZE as useful subclasses for AUTOMOBILE

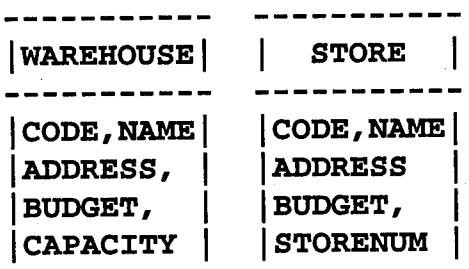


However, some relationships cannot be represented using the above steps. For example, when you need to define two (or more) kinds of classes with similar but not identical properties, and neither should be a subclass of the other, do:

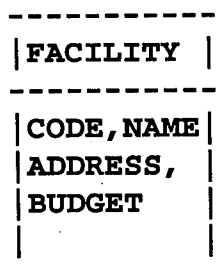
- abstract the common properties of the two (or more) kinds of classes to obtain a new class
- designate the new class as a superclass
- make the classes in question subclasses for the new superclass

For example, suppose we start with the classes

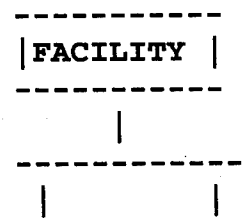
STORE and WAREHOUSE



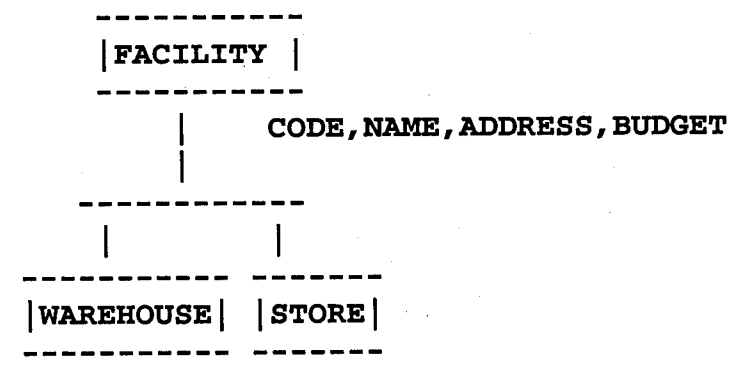
Step1: An abstraction of the common properties of WAREHOUSE and STORE could be FACILITY



Step2: Satisfied with the analysis in Step1, FACILITY is designated the superclass.



Step3: WAREHOUSE and STORE are specified as subclasses of FACILITY



CAPACITY STORENUM

A superclass need not provide direct data management functionality; its sole purpose might be to organize your representation of the world correctly, making the class hierarchy understandable, and leaving a framework for future expansion.

Besides the user-defined classes of the designer's model, Gemstone has a builtin class hierarchy. The builtin hierarchy consists of very basic and important classes such as numbers, arrays, and sets complete with their data structures and behaviors. Figure 32 displays a small subset of the builtin hierarchy (Gemstone 1994).

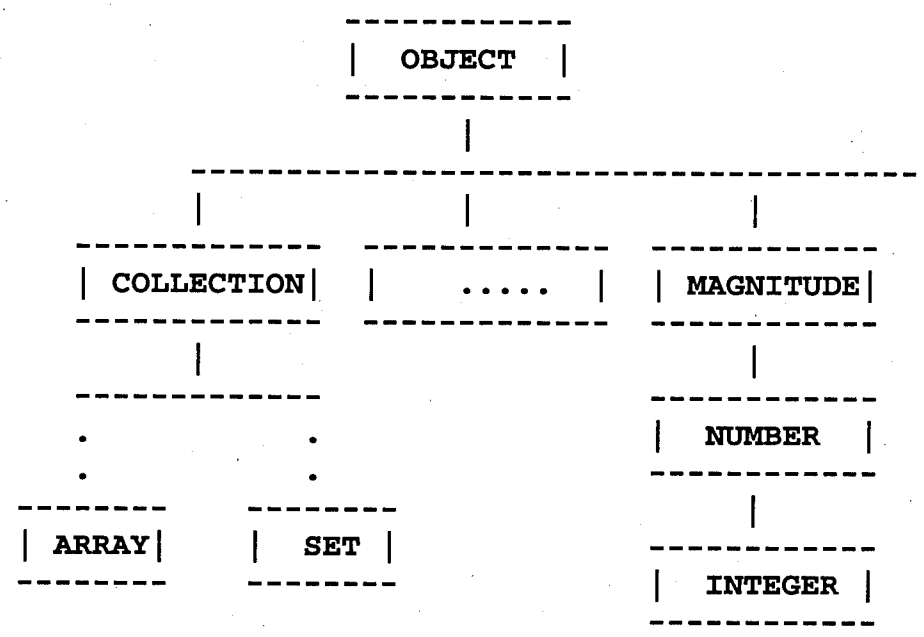


Figure 32. Subset of the Gemstone class hierarchy

Recall that we examined the INTEGER class in Section 2.2. Here we see that in Gemstone and in other object DBMSs we do not have to create the INTEGER class since it is a builtin class. You can add new classes almost anywhere in the hierarchy to take advantage of the data structures and methods that have already been defined. For example, making EMPLOYEE class a subclass of SET, in Figure 33, makes it possible to model EMPLOYEE as a relation wherein the tuples are unique and ordering of tuples is immaterial.

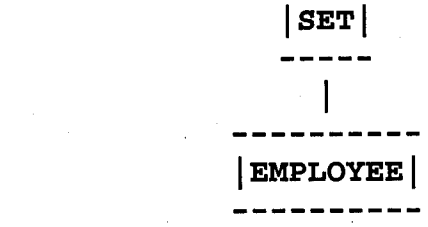


Figure 33. Set of employees

This is done as follows:

```
Object subclass: 'EMPLOYEE'
     instVarNames: #('NAME' 'JOB' 'AGE' 'ADDRESS')
     constraints: ...

Set subclass: 'SetOfEmployees'
     instVarNames: #()
     constraints: #(EMPLOYEE)
```

Another situation which uses the SET class later in this chapter is the hierarchy in Figure 34.

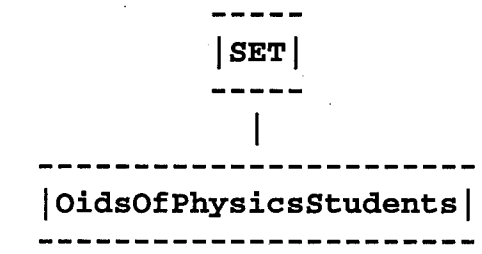


Figure 34. Set of OIDs of Physics students

The class OidsOfPhysicsStudents inherits set properties such as uniqueness and absence of ordering.

Much of the benefit of object orientation in database systems is derived from both structural and behavioral inheritance. Thus designing inheritance relationships is extremely important and warrants careful analysis.

4.2 Interclass Relationship

Relationships in the ER model are expressed explicitly using a diamond symbol between related entity types. The representation of a relationship in the relational model is fairly implicit and is achieved by embedding the primary key of a target relation into a related relation. In the latter, the inherited attribute is known as a foreign key. Following this trend, in the object model, a related object is entirely embedded in a target object. In an OO diagram the class of interest is represented by a rectangle into which may be embedded zero, one, or many related classes.

For example, the relationship which asserts that a student is associated with a department is diagrammed in Figure 35.

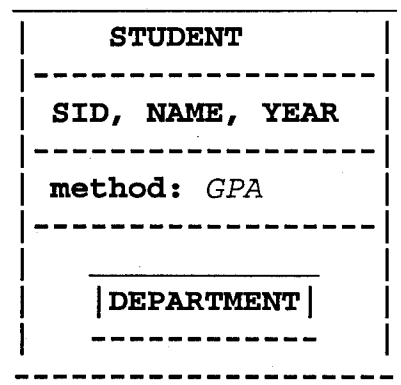


Figure 35. STUDENT class with relationship to DEPARTMENT

Since a relationship is mutual we could also have the relationship in Figure 36.

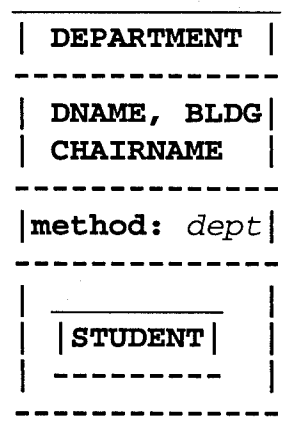


Figure 36. DEPARTMENT class with relationship to STUDENT

4.2.1 Connectivity

Connectivity of the relationship between classes is useful in the OO model and the possible ratios are 1:1, 1:n, m:n (Batini, Ceri, and Navathe 1992). Consider the policy:

> A student is in one department

This policy is diagrammed in Figure 37 as a STUDENT class in a one-relationship with DEPARTMENT.

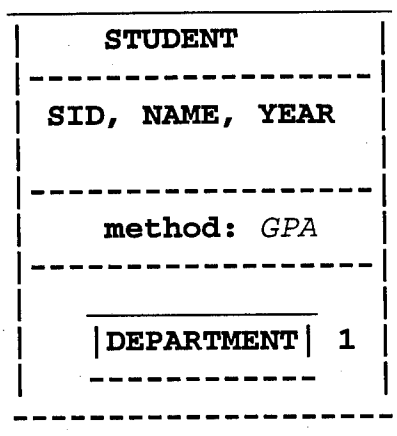


Figure 37. A student object belongs to one department

Consider the policy

A department has many students

This policy is diagrammed in Figure 38 as a DEPARTMENT class in a many-relationship with STUDENT.

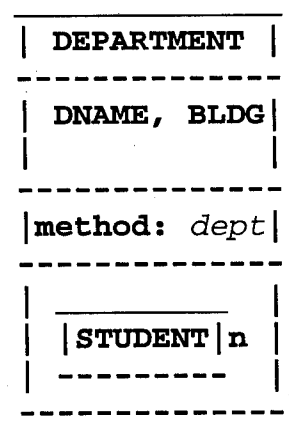


Figure 38. A department object has many students

We shall see that a many-relationship, such as DEPARTMENT has with STUDENT, is implemented by introducing a SET subclass.

4.2.2 Membership

The concept of membership of objects in a relationship is also relevant in OO models and may be shown in the object

diagram. If we restate the above policy to account for undeclared majors, we have

A student is in zero or one department

then membership of student in department is optional and this is denoted in Figure 39 by the embedded single-sided rectangle representing the DEPARTMENT class.

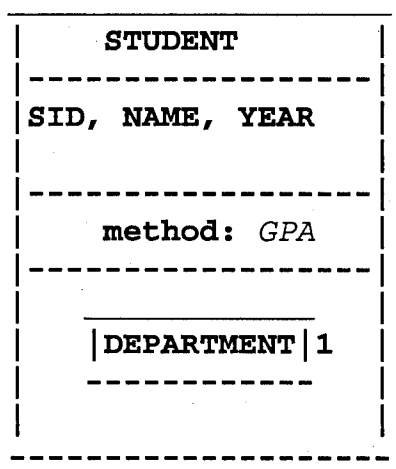
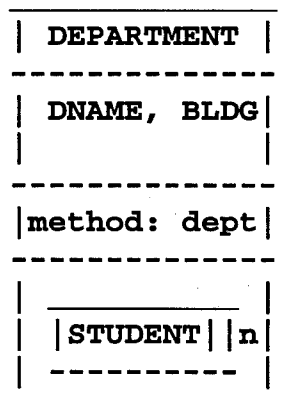


Figure 39. STUDENT class with optional membership in a relationship

On the other hand, the policy

A department has one or more students

demands a mandatory membership for a department. That is, a department must have at least one student. This is denoted by the embedded double-sided rectangle representing the STUDENT class in Figure 40.



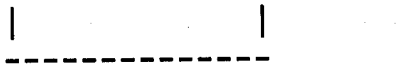


Figure 40. A DEPARTMENT with mandatory membership in a relationship

Despite the relevance of optional memberships in object databases, we will assume mandatory memberships for all relationships in this tutorial and ignore the distinction between the two kinds of membership and their notations.

4.2.3 Representation of interclass relationships

Interclass relationships are implemented with instance variables. Thus the embedded DEPARTMENT class in Figure 41 is replaced with DEPT instance variable as shown in Figure 42.

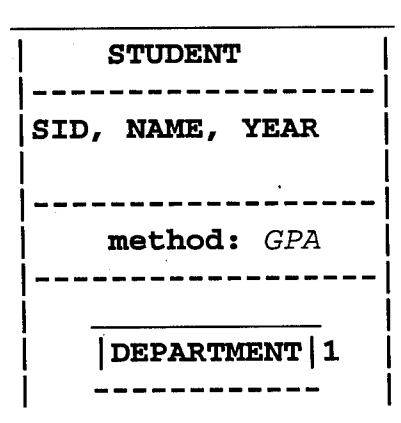


Figure 41. STUDENT-DEPARTMENT relationship

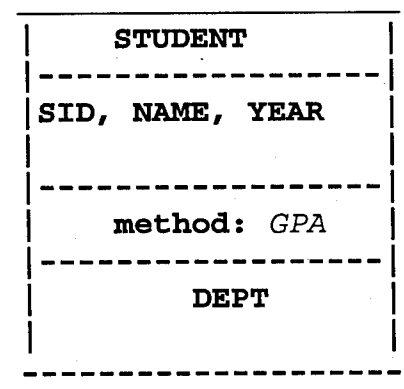


Figure 42. Instance variable representation of DEPARTMENT

While instance variables SID, NAME, YEAR of STUDENT assume primitive data values, DEPT which represents a related object assumes an OID as its value. STUDENT's relationship with DEPARTMENT, a one-relationship, is implemented as an instance variable whose value is a reference to the appropriate DEPARTMENT object. An object state of STUDENT is shown in Figure 43.

STUDENT class

instvar	instance
SID	555 23 6657
NAME	Smith
YEAR	senior
DEPT	OidOfDept

Figure 43. An object state of STUDENT

Now consider DEPARTMENT's relationship with STUDENT in Figure 44.

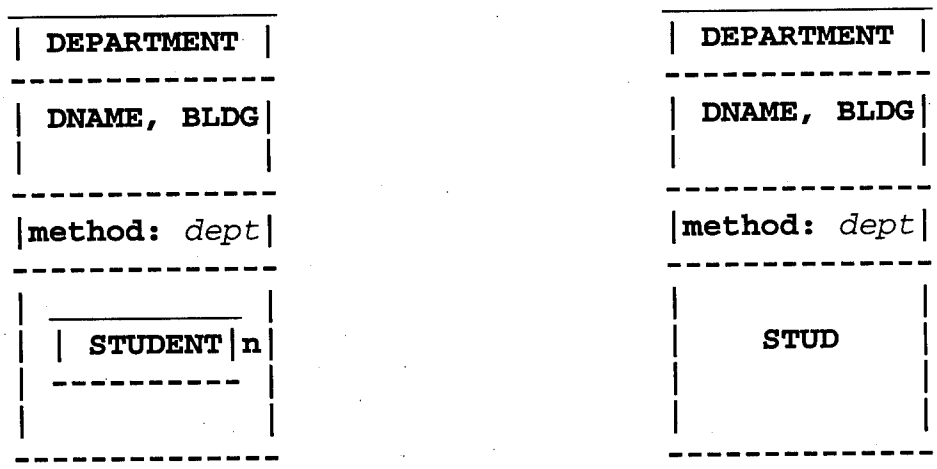


Figure 44. Instance variable representation of STUDENT

DEPARTMENT's relationship with STUDENT, a many-relationship, is implemented as an instance variable, STUD, whose value is an OID that references a set of students, Figures 44 and 45. In this case there are more than one student in each department and a set is used for the implementation.

DEPARTMENT Class

instvar	instance
DNAME	physics
BLDG	JAP
STUD	OidOfSetOfStudents

Figure 45. An object state of DEPARTMENT

More on the implementation details of relationships using instance variables will be said in Section 4.2.5.1 and Appendix C.

Clearly the value in an instance variable may be a primitive or an OID that references object(s) of arbitrary complexity, enabling object DBMSs to provide support for complex objects.

In the next subsection, we digress slightly but appropriately to take a brief look at how object diagrams may be developed. This will give us a sense of how real-world interactions may be captured as interclass relationships.

4.2.4 Mapping an ER model to an object diagram

Disregarding behavior for a moment, the concept of class is analogous to the combined concepts of entity and relationship types. Thus it seems appropriate to capture relationships for an object model through the use of ER techniques. This prospect is particularly appealing since the ER model has withstood the test of time, is well understood, and is widely used in the data management industry. In the first phase, a conceptual ER model is obtained. Then using a technique that is similar to the process of translating a conceptual model to a logical design for a relational DBMS, the conceptual model is mapped to an object diagram. For the purpose of mapping ER diagrams to object diagrams, a comprehensive set of mapping rules needs to be derived. Here, we show two such mapping rules: one for 1:n, the other for m:n.

Case 1. Binary 1:n ER Diagram

A mandatory ER diagram with 1:n connectivity is shown in Figure 46. Entity type P with attributes attr(P) has a one-to-many relationship R with entity type Q which has attributes attr(Q). This ER diagram maps to the object diagram in Figure 47.

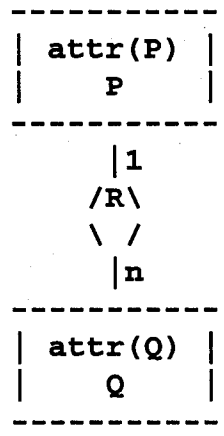
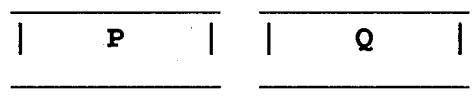


Figure 46. 1:n ER diagram of entity types P and Q



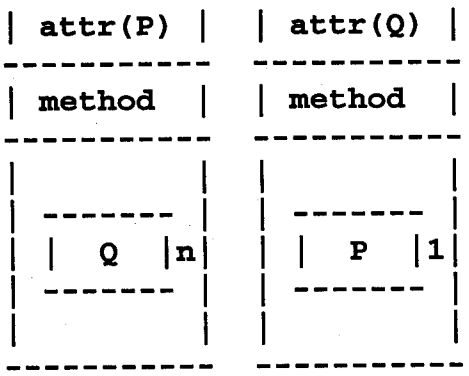


Figure 47. Object diagram of classes P and Q

Case 2. Binary m:n ER Diagram

The binary ER diagram of entity types P and Q in Figure 48 maps to the object diagram in Figure 49. Observe that the relationship type R becomes a class R with n:1 relationships to classes P and Q. R is known as an intersection class.

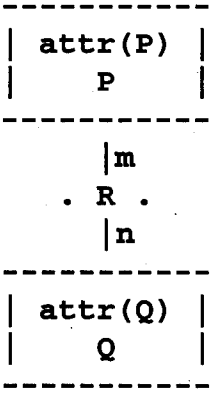


Figure 48. m:n ER diagram of entity types P and Q

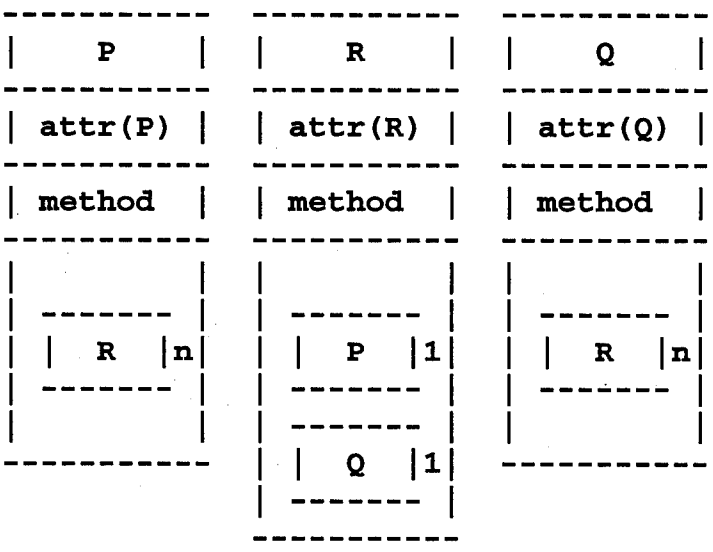


Figure 49. Object diagram of classes P, Q, and R

With a full set of mapping rules, it is possible to draw an ER diagram which may subsequently be mapped to an object diagram.
If this is done, the designer could define methods for the resulting classes.

4.2.5 Instance diagram

An instance diagram illustrates logically how objects of an object diagram are stored in a database. It is insightful to draw instance diagrams for given object diagrams. For example, an instance diagram may serve as a useful guide for writing the code to populate the object database. We will discuss the instance diagram for two situations:

Case 1. A 1:n relationship
Case 2. A m:n relationship.

4.2.5.1 Case 1 - Instance diagram for 1:n relationship

Consider the DEPARTMENT-STUDENT class relationship diagram in Figure 50.

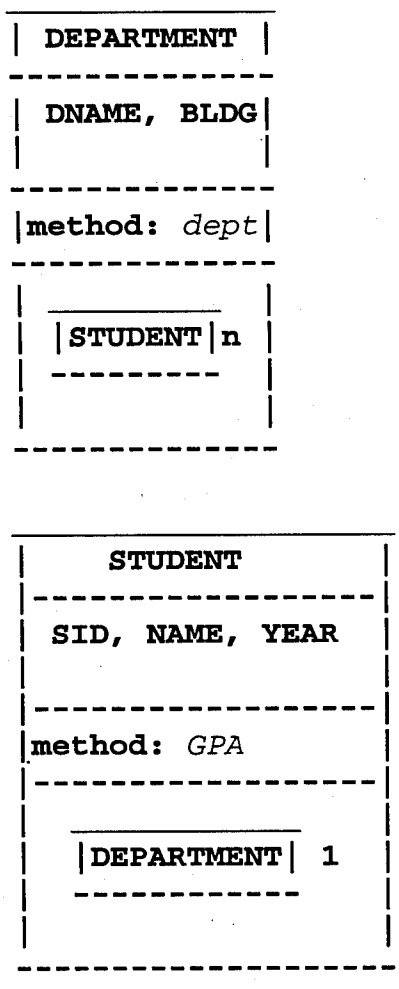


Figure 50. DEPARTMENT-STUDENT relationship

The corresponding instance diagram is given in Figure 51. Instance diagrams make heavy use of OIDs, in part, to represent relationships through references to OIDs. For example, Physics department with OID=D001 contains the data value OID=cs102 (shown in square brackets) for instance variable STUD. This OID references a set of student OIDs. These particular students have a relationship with the Physics department. The individual OIDs of the set reference the actual STUDENT objects. Each DEPARTMENT instance has an instance variable STUD which references the department's own set of student OIDs.

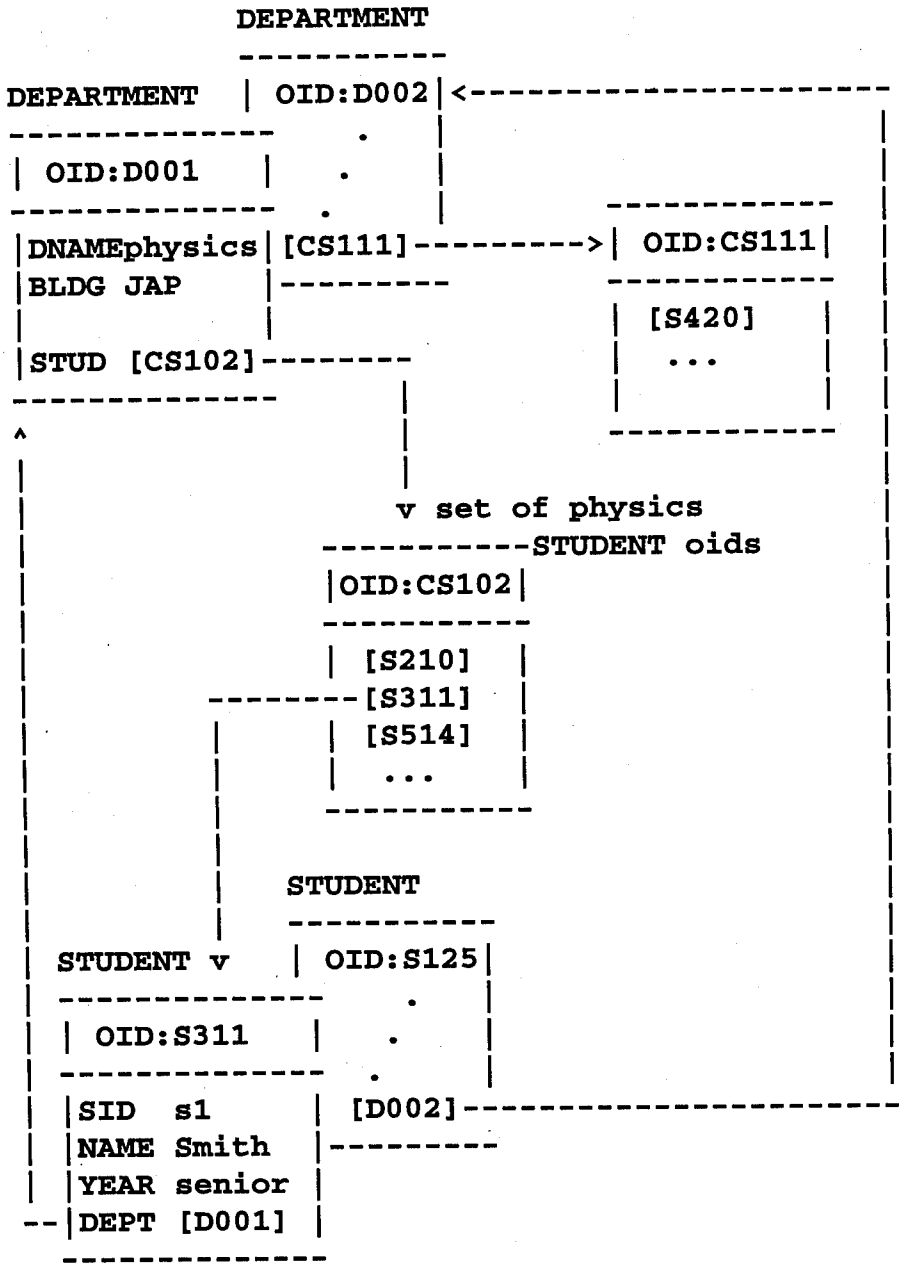


Figure 51. DEPARTMENT-STUDENT instance diagram

Relationships are implemented by an object referencing a related object using the object's OID. The object does this with an instance variable whose value is the related object's OID. In the following we give a brief overview of how OIDs are handled by examining student Smith's relationship with a department. The tabular equivalent of the STUDENT class and its relationship to the DEPARTMENT class is shown in Figure 52. Note that in this tabular representation, the value of DEPT is not primitive but

rather a complete object. In an OO system, this object is represented by its OID. Relational systems do not support this kind of table.

STUDENT Class

SID	NAME	YEAR	DEPT
s1	Smith	senior	physics
			JAP
			STUD
s2	Clark	junior	physics
			JAP
			STUD

Figure 52. Tabular display of STUDENT class

In the following, recall that the message *new* causes new objects of a class to be created. At the moment of its creation an object is assigned an OID. The following code creates an instance, Smith, for the STUDENT class.

```
| S |                    "declare S to be a temporary variable"

S := (STUDENT new)
     SID:'s1'; 'NAME:'Smith'; YEAR:'senior'; DEPT:'D001'
```

As soon as it is created, the object representing STUDENT Smith is given an OID. With reference to Figure 51, the OID with value S311 is immediately assigned to the temporary variable S, thus S=S311. In a program, S represents the Smith instance. The DEPT instance variable contains the OID of Smith's department, Physics. Note that STUDENT has a one-relationship with DEPARTMENT as shown in Figure 50. In general, one-relationships are handled as described above.

The situation is somewhat different in a many-relationship. In this case we track the Physics department's relationship with its students. The tabular equivalent of the DEPARTMENT Class and its relationship to the STUDENT class is shown in Figure 53. Note that in this tabular representation, the value of STUD is not primitive but rather several complete objects. In an OO system, the OID of this set of objects is the value of STUD. Relational systems do not support this kind of table.

DEPARTMENT Class

DNAME	BLDG	STUD

```
physics  JAP     s1      s2      s3      ...
                 Smith   Clark   Dixon
                 Senior  Junior  Junior
                 DEPT    DEPT    DEPT
geology  DSN     s7      s4      ...
                 Blake   Johnson
                 Senior  Junior
                 DEPT    DEPT
```

Figure 53. Tabular display of DEPARTMENT class

Recall the connectivity of the relationship in Figure 50:

A DEPARTMENT has several students.

And consider the creation of a Physics DEPARTMENT instance.

```
| D |                "declare D to be a temporary variable"

D := (DEPARTMENT new)
     DNAME:'physics'; BLDG:'JAP'; STUD:'cs102'
```

First you create a set of student OIDs for each department by defining the former as a subclass of the builtin Set class. These sets are objects in their own right. Therefore each has an OID, say, cs102 and cs111 in Figure 51. Each set is referenced by the appropriate DEPARTMENT instance through its STUD instance variable. In particular, instance variable STUD of the Physics department references the set object cs102. In the diagram, the object with OID=cs102 contains the set of OIDs s210,s311, s514. In turn, each of these OIDs references a STUDENT object. Thus starting from DEPARTMENT, STUD references the set cs102 which, in turn, references Physics student objects.

It should be clear from the foregoing that the notation for containment refers to an OID and not to the class itself. For example the diagram of Figure 54 should be understood to mean that each department object has an instance variable STUD, say, whose value is an OID of a set of student OIDs.

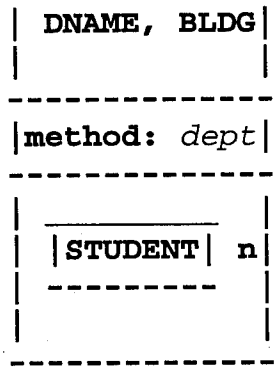
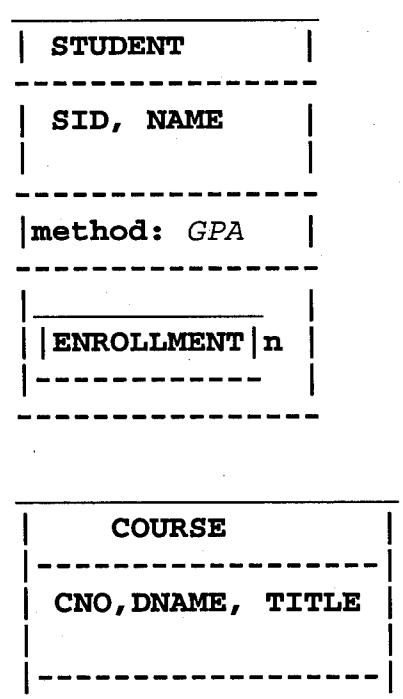


Figure 54. DEPARTMENT with a many-relationship

Appendix C contains a program that discusses and demonstrates the implementation of interclass relationships.

4.2.5.2 Case 2 - Instance diagram for m:n relationship

The treatment of m:n relationship is a little different from the 1:n and is reminiscent of the Codasyl m:n. First an intersection class is introduced following the mapping rule of Section 4.2.4. A STUDENT-COURSE relationship is used to illustrate the m:n situation, Figure 55.



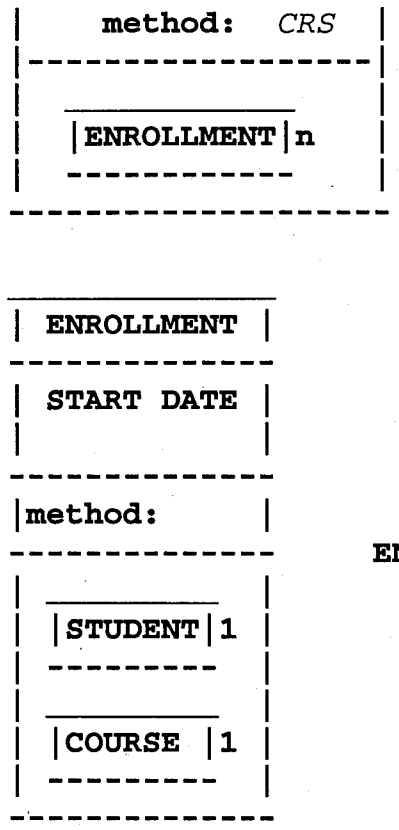
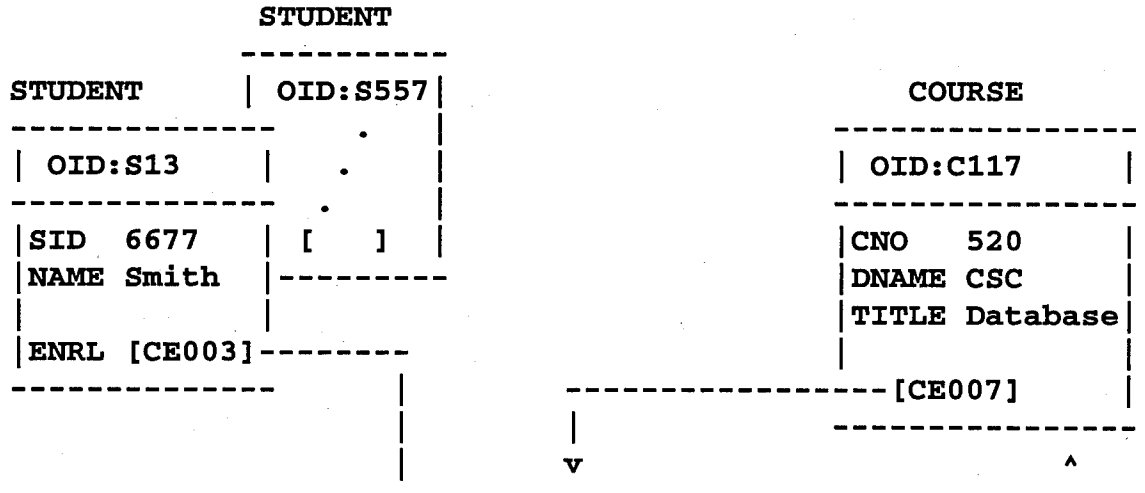


Figure 55. m:n STUDENT-COURSE relationship

Like relationship types in the ER model, the intersection class can acquire its own instance variables. For example, note the inclusion of a new instance variable, START DATE, in the ENROLLMENT class. The instance diagram is given in Figure 56. Navigation through this instance diagram follows the same principles as those of the previous section.



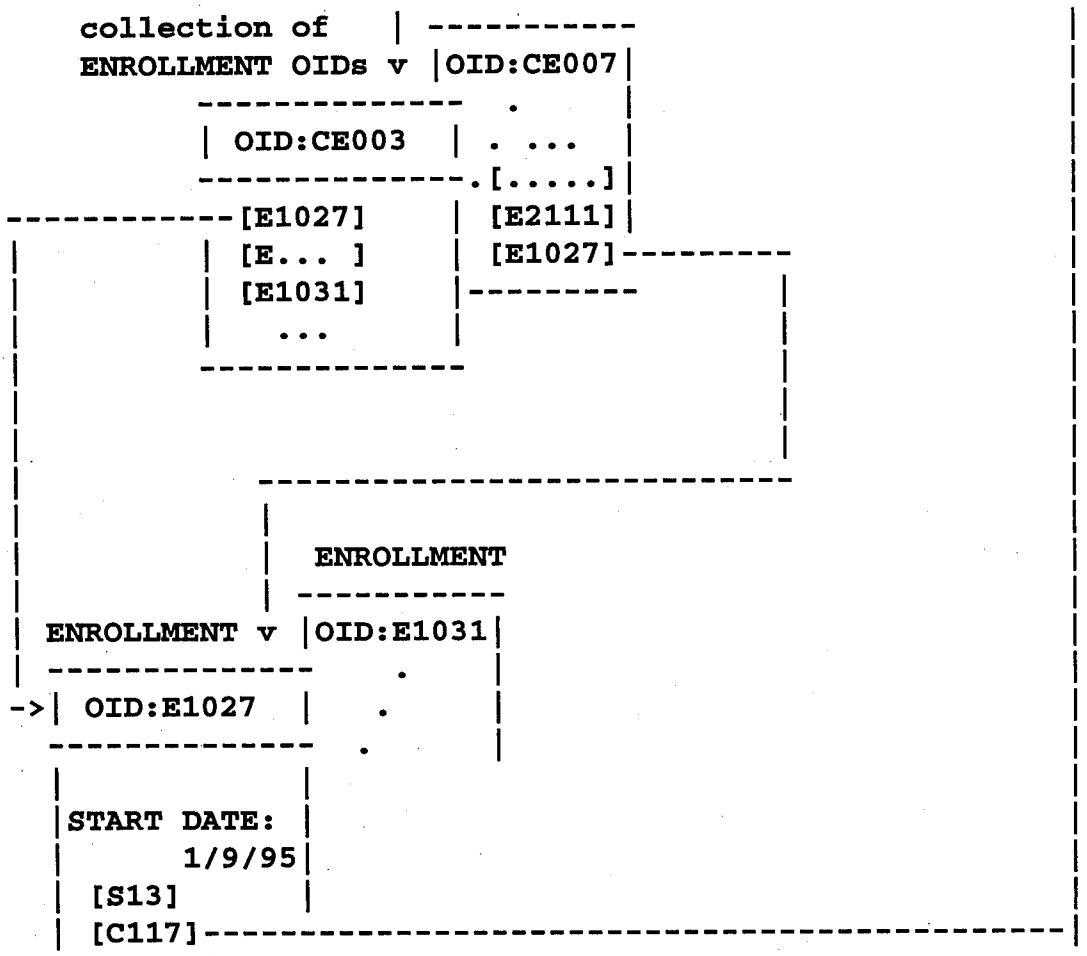


Figure 56. STUDENT-COURSE instance diagram

4.3 Aggregation Relationship

Composite instance variables, first discussed in Chapter 2, constitute an important kind of object interaction. Consider the STUDENT class in Figure 57.

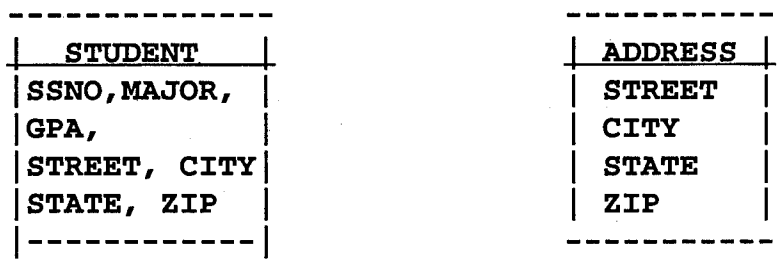
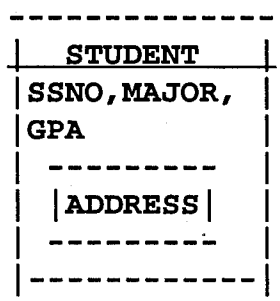


Figure 57. A composite instance variable

Recognizing that STREET, CITY, STATE, ZIP make up a composite instance variable, ADDRESS, we could create the new class, ADDRESS, with instance variables STREET, CITY, STATE, ZIP as shown in Figure 57. Clearly the ADDRESS class has a relationship with the STUDENT class. Thus, the STUDENT-ADDRESS object diagram may be presented as in Figure 58.



or equivalently

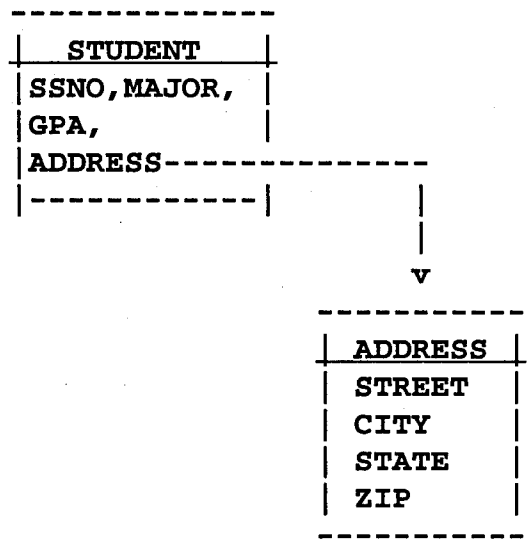


Figure 58. STUDENT-ADDRESS relationship

The second diagram in Figure 58 is preferable when the motivation for aggregation is referential object sharing by several classes. Now the connectivity of the relationship may be determined to obtain, say, the diagram of Figure 59.

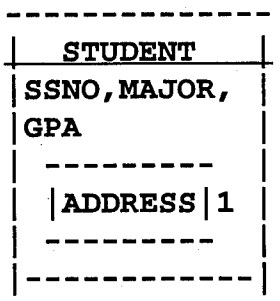


Figure 59. Connectivity of STUDENT-ADDRESS relationship

The instance diagram for this model is given in Figure 60. The two students, s19 and s23, live at the same ADDRESS and share the same ADDRESS object, rather than referencing two different ADDRESS objects with equal object states. This is known as referential object sharing which is advantageous in updates of ADDRESS.

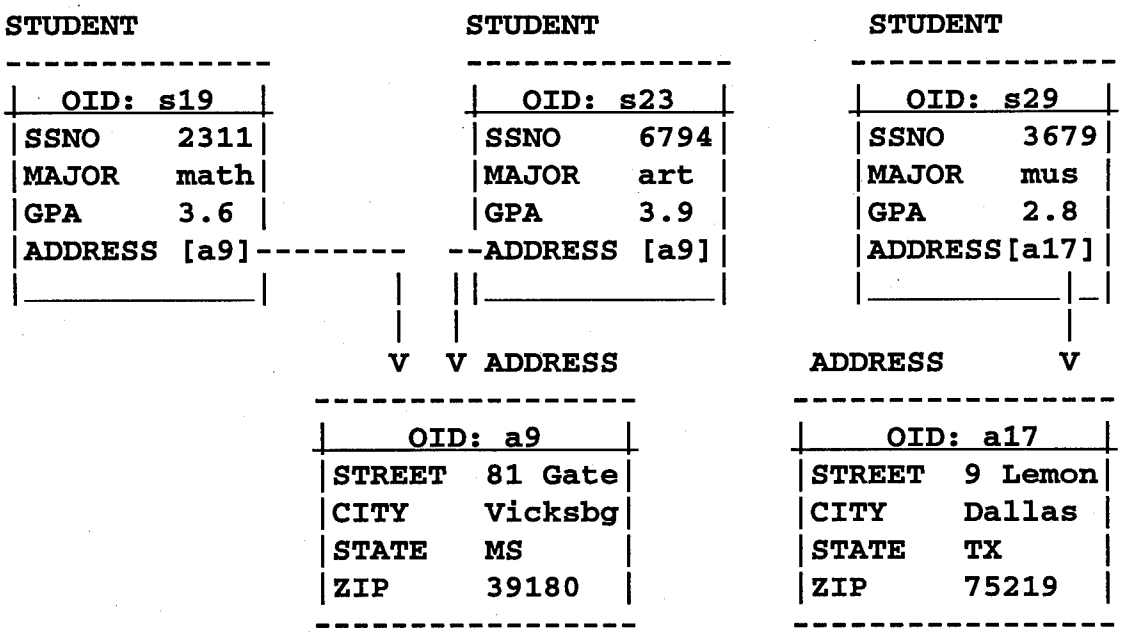
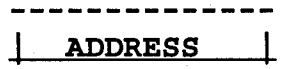


Figure 60. Instance diagram of STUDENT-ADDRESS relationship

The above analysis is based upon STUDENT's relationship to ADDRESS. Next, the designer determines whether or not the reverse relationship from ADDRESS to STUDENT is relevant. If so, then its connectivity is determined and analyzed in the usual way to obtain, for example, Figure 61. That is, an ADDRESS may belong to many students.



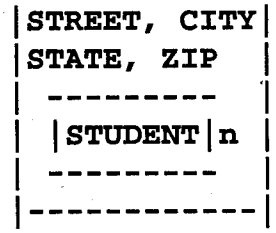


Figure 61. ADDRESS-STUDENT relationship

As a modeling construct, factoring out composite instance variables reduces to relationships between classes. This kind of relationship is called an aggregation relationship (Bertino and Martino 1991). The analysis in Section 4.2, including instance diagramming and implementation considerations, are applicable to aggregation relationships.

Assignment:

Draw an instance diagram for a m:n relationship between two given classes.

5. OBJECT MODEL

Like in previous database approaches, data intended for use in an object database must undergo modeling. The object model should support the modeling of object structures, behaviors, and relationships. While the goal of semantic models is to provide mechanisms for structural and relationship abstraction only, that of object models is to provide mechanisms for structural and behavioral abstraction as well as object interaction. A general strategy to represent these object features is as follows. Because of the primacy of inheritance in object databases, representation of the hierarchy should provide the structural underpinning for an object model. As we have seen in Chapters 2, 3, and 4, the construction of hierarchies is driven by structural and behavioral considerations. The designer may choose to inherit class hierarchies from an ER model of the enterprise since the ER approach supports class hierarchies. However, ER class hierarchies are not normally based upon behavior and therefore the designer should take behavior into consideration before adopting such class hierarchies. In any case, the design of class hierarchies is the first task toward drawing an object model. Relationships, including those originating from composite instance variables, may then be superimposed on the class hierarchies. The analysis on relationships and the process of deriving them from an ER model of the enterprise was presented in Chapter 4. The next several sections summarize the various elements of the object model.

5.1 Representation of Classes in the Object Model

In keeping with the notation introduced earlier, we will represent a class by a rectangle labeled with a descriptive name.

5.2 Representation of Instance Variables in the Object Model

The approach recommended for including instance variables in the object model is to enumerate them by class under the class heading. Because there is generally a relatively large number of instance variables, it may not be

convenient to write them in the diagram itself. Rather, the list of instance variables may be given on a separate medium, such as a system dictionary, which is referenced by the object diagram.

5.3 Representation of Methods in the Object Model

The approach advocated is to provide the names and functionality of methods for a class in the class protocol. As in the case of instance variables, protocols may be provided on a separate medium, such as a system dictionary.

5.4 Representation of Inheritance Relationships in the Object Model

We have, in our discussions, represented inheritance relationships by straight line segments. We formally adopt this practice for the object model.

5.5 Representation of Interclass Relationships in the Object Model

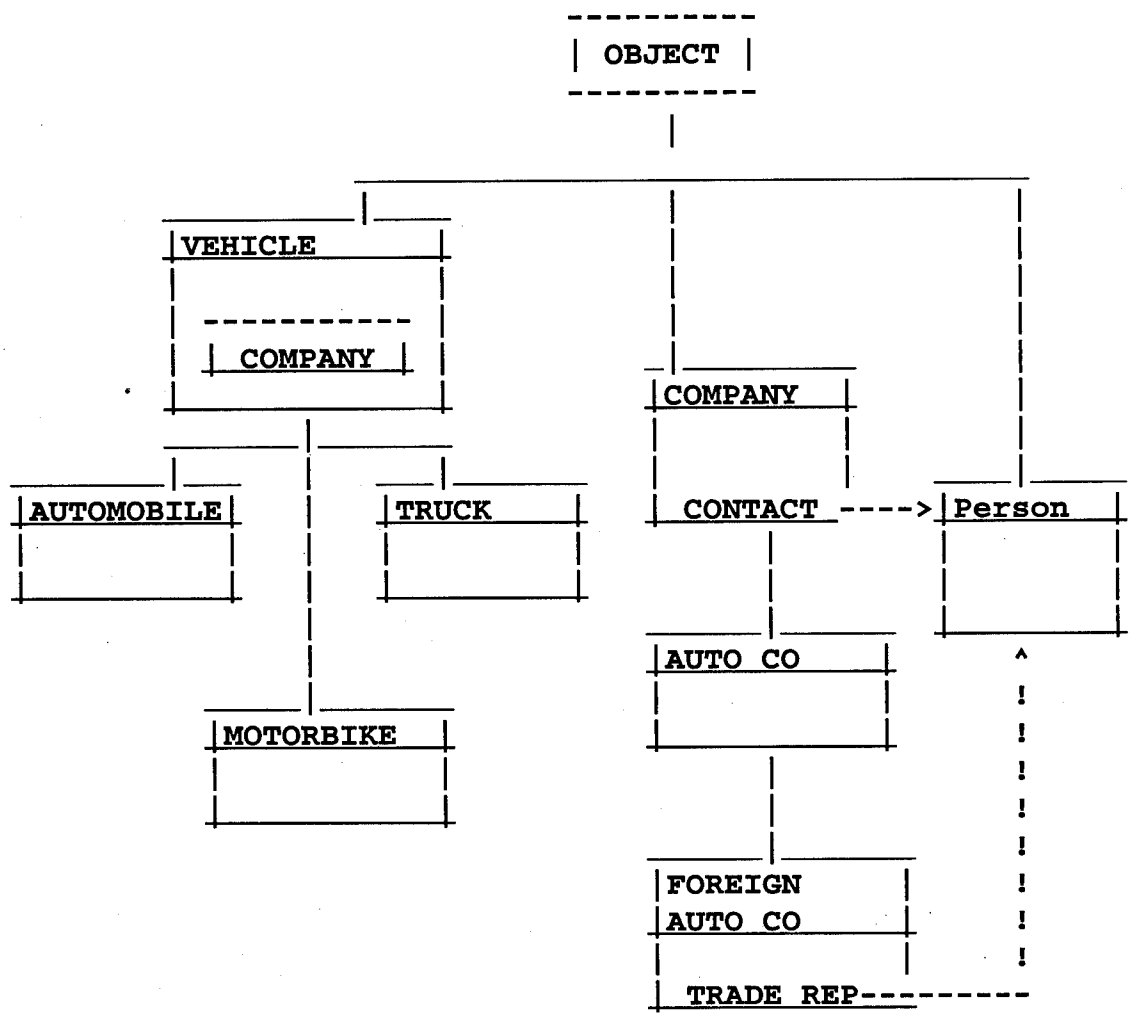
A relationship between classes is represented by embedding one class in the other. For example, if class A has a relationship with class B, then class B is embedded in class A following the containment notation used in Chapter 4.

5.6 Representation of Aggregation Relationships in the Object Model

An aggregation relationship may be denoted either by containment or by an arrow from the parent class to the derived class. When the latter notation is used, the arrows are relationships and should not be confused with the pointers of instance diagrams.

5.7 Object Model Example

Figure 62 is an object model of a manufacturing COMPANY, adapted from McFadden and Hoffer (1991). The diagram follows the conventions discussed above.



_____ inheritance relationship
------> aggregation relationship

Instance Variables

VEHICLE	COMPANY	AUTOMOBILE	TRUCK
SERIAL#	NAME	TRUNK CAPAC	CARGO
SPACE			
WEIGHT	LOCATION		
WHEELBASE	CONTACT		

MOTORBIKE	AUTO COMPANY	FOREIGN AUTO	PERSON
SEAT STYLE	ALLOCATION	IMPORT LIMIT	NAME
	TOTAL SALES	TRADE REP	DOB

Protocols

VEHICLE Initialization/Updating Protocol

Message pattern	Usage
SERIAL#: aNumber	Initializes SERIAL# to aNumber.
WEIGHT: aNumber	Initializes WEIGHT to aNumber.
WHEELBASE: aNumber	Initializes WHEELBASE to aNumber.

VEHICLE Retrieval Protocol

Message pattern	Usage
SERIAL#	Retrieves value of SERIAL#.
WEIGHT	Retrieves value of WEIGHT.
WHEELBASE	Retrieves value of WHEELBASE.

and so on.

Figure 62. Object model of an auto MFG company

5.8 Conclusion

An object model and the corresponding instance diagram embody all the concepts - object identity, structure, behavior, interaction, inheritance, and encapsulation - of the object-oriented paradigm. With an object model, database implementation may begin and Appendixes B and C give examples of this process. The example in Appendix B contains inheritance but no interclass relationships and

the implementation program is relatively simple. However, the example in Appendix C has both inheritance and interclass relationships, making the implementation program quite intricate. Data definition, database population, and the query capability in Gemstone are demonstrated in the examples.

As with many other areas in computer science, true understanding of a novel concept comes only from hands-on work. In the case of object-oriented database systems, this means constructing simple object models and writing programs to implement and process them. This tutorial gives the fundamentals to tackle these tasks. Appendix A is a user guide with a guest account number which makes available the Gemstone object DBMS at Jackson State University for implementing practice object databases.

References

Batini, C., Ceri, S., and Navathe, S.B. (1992). *Conceptual database design: an entity-relationship approach.* **Benjamin-Cummings, Redwood City, CA, 30-48.**

Bertino, E., and Martino, L. (1991). "Object-oriented database management systems: Concepts and issues," *IEEE Computer* **4, 33-47.**

Cattell, R.G.G. (1994). *Object data management.* **Addison-Wesley, Reading, Mass., 104-135.**

Date, C.J. (1995). *An introduction to database systems.* **6th ed., Addison-Wesley, Reading, Mass., 626-709.**

Gemstone Corporation. (1994). *Gemstone programming guide.* **Gemstone Corporation, Beaverton, Oregon.**

Goldberg, A., and Robson, D. (1983). *Smalltalk-80: the language and its implementation.* **Addison-Wesley, Reading, Mass.**

Khoshafian, S.N., and Copeland, G.P. (1986). "Object iden tity," *ACM proceedings of the conference on OOPSLA.* **New York, 406-416.**

Loomis, M.E.S. (1995). *Object databases: the essentials.* **Addison-Wesley, Reading, Mass., 40-45.**

McFadden, F.R. and Hoffer, J.A. (1991). "Data concepts and modeling." *Database management.* **Benjamin/Cummings, Redwood City, CA, 121-125.**

Rob, P., and Coronel, C. (1993). *Database systems: design, implementation, and management.* **Wadsworth, Belmont, CA, 417-474.**

Appendix A

Gemstone User Guide

PART 1) TO ACCESS THE NETWORK

```
F> telnet stallion.jsums.edu

LOGIN:    gemusr1
PASSWORD: usrgem1

(enter 2 for terminal type 2)

$
```

THE FOLLOWING COMMAND RETURNS 2 FILENAMES WHICH INDICATE THAT GEMSTONE IS UP

```
$ stonelist

GEMSERVER40
NETLDI40
```

PART 2) TO LOG ON

```
$ topaz1

TOPAZ > set gemstone gemserver40 user gemusr1 password
gemstone

TOPAZ > login

TOPAZ 1> (see Part 4 for entering a session)
```

PART 3) TO LOG OFF

```
TOPAZ 1> exit

$ exit

F>
```

PART 4) A GEMSTONE SESSION

NOTE THAT GEMSTONE IS CASE SENSITIVE

TO DEFINE A CLASS.

```
TOPAZ 1> printit
Object subclass: 'Animal'
     instVarNames:  #('NAME' 'favoritefood' 'habitat')
     inDictionary:  UserGlobals
     constraints:   #[
                     [#NAME, String],
                     [#favoritefood, String],
                     [#habitat, String]
                     ].
%

TO CREATE AN INSTANCE OF, SAY, ANIMAL

TOPAZ  1> printit
UserGlobals at: #aDog put: animal new.
"This is equivalent to |aDog|  aDog := animal new."
%

TO RETRIEVE THE VALUE OF aDOG'S NAME

TOPAZ 1> printit
aDog NAME
%

TO CREATE A METHOD, SAY, FOR ANIMAL

TOPAZ 1> set class animal

TOPAZ 1> method: ^
NAME: aNAME
     NAME := aNAME
%


PART 5)   TO EDIT THE PREVIOUS RUN USING vi

TOPAZ 1> set editorNAME vi
TOPAZ 1> edit last

PART 6)   INPUT FROM A FILE

YOU CAN DEVELOP YOUR SCRIPT WITH vi AND INPUT THE
FILE INTO TOPAZ

TOPAZ 1> spawn
```

```
$ vi fileNAME

(type up the script as in Part 4.)

$
```

TO RETURN TO TOPAZ FOR EXECUTION

$ ^d

TOPAZ 1> input $HOME/fileNAME

PART 7) TO CAPTURE YOUR TOPAZ SESSION IN A FILE

TO CAPTURE YOUR TOPAZ SESSION IN A FILE, SAY,
ANIMALTEST.LOG FOR THE PURPOSE OF DEBUGGING.

TOPAZ 1> output push animaltest.log

START YOUR SESSION NOW. WHEN THRU, SPAWN TO vi TO
SEE THE SESSION in ANIMALTEST.LOG.

Appendix B

Inheritance Relationship Implementation

An Employee Database

The implementation of an object model that consists solely of hierarchical relationships is relatively simple. About as simple as a relational system implementation. The following is the implementation of an employee database. The example illustrates initializing, updating, and retrieving methods; structural and behavioral inheritance; and polymorphism.

Object Model for an Employee Database

The data model consists of three employee classes: SECRETARY, SCIENTIST, ENGINEER. The three classes are grouped into a class EMPLOYEE which contains the shared characteristics of the subclasses SECRETARY, SCIENTIST, ENGINEER. The class hierarchy is given in Figure B1.

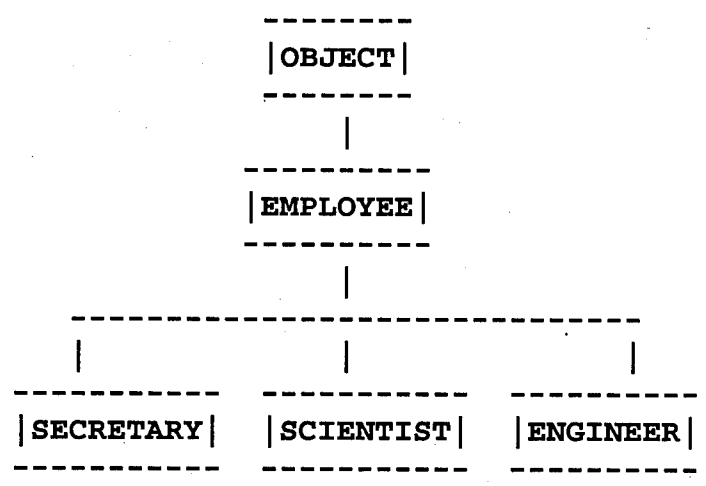


Figure B1. Employee Class Hierarchy

Gemstone Data Definition

In Gemstone's OO environment, all classes are derived from the OBJECT superclass and, for that reason, inherit all of its characteristics. Note that class definitions are made

in a hierarchical sequence. The following example creates EMPLOYEE as a subclass of OBJECT:

```
printit
Object subclass: 'employee'
     instVarNames: #('ssno' 'name' 'expertise' 'salary')
     inDictionary: UserGlobals
     constraints: #[
               #[#ssno, String],
               #[#name, String],
               #[#expertise, String],
               #[#salary, Integer]
                ]
%
```

Naturally, a subclass is likely to have characteristics that distinguish it from its superclass. For example,

the SECRETARY subclass has a unique numserviced - number of employees supported;
the SCIENTIST subclass has a unique numpublished - number of papers published;
the ENGINEER subclass has a unique numprojects - number of projects undertaken.

The OO environment makes it easy to add such specializing instance variables once the subclass is derived from its superclass. The following lines create the subclasses SECRETARY, SCIENTIST, ENGINEER:

```
printit
employee subclass: 'secretary'
     instVarNames: #('numserviced')
     inDictionary: UserGlobals
%
printit
employee subclass: 'scientist'
     instVarNames: #('numpublished')
     inDictionary: UserGlobals
%
printit
employee subclass: 'engineer'
     instVarNames: #('numprojects')
     inDictionary: UserGlobals
%
```

Methods for Populating/Updating and Retrieving

Each method is a code that defines a specific operation that we want the object to perform. Each method has a name to identify it. A class provides storage for all the methods that describe the behavior of the objects in that class. The methods for the EMPLOYEE class are derived next.

```
set class employee
method: ^
ssno: aString

"Modify the value of the instance variable 'ssno'."
     ssno := aString
%
method: ^
name: aString

"Modify the value of the instance variable 'name'."
     name := aString
%
method: ^
expertise: aString

"Modify the value of the instance variable 'expertise'."
     expertise := aString
%
method: ^
salary: anInteger

"Modify the value of the instance variable 'salary'."
     salary := anInteger
%
method: ^
salary

"Return the monthly salary of employee."
     ^salary
%
method: ^
raise

     ^(salary*0.05)
%
```

```
set class secretary
method: ^
numserviced: anInteger

"Modify the value of the instance variable 'numserviced'."
     numserviced := anInteger
%
method: ^
total

"Return a secretary's number of employees serviced."
     ^numserviced
%
set class scientist
method: ^
numpublished: anInteger

"Modify the value of the instance variable 'numpublished'."
     numpublished := anInteger
%
method: ^
total

"Return a scientist's number of publications."
     ^numpublished
%
set class engineer
method: ^
numprojects: anInteger

"Modify the value of the instance variable 'numprojects'."
     numprojects := anInteger
%
method: ^
total

"Return an engineer's number of projects."
     ^numprojects
%
```

Instantiation

Next we will create some instances of the subclasses we have already defined.

```
printit

UserGlobals at: #edna put: secretary new.
%
printit

edna ssno: '555 23 9946'; name: 'edna shields'; expertise:
'secretary'; salary: 19000; numserviced:
15
%

printit

UserGlobals at: #bob put: scientist new.
%
printit

bob ssno: '666 73 6645'; name: 'bob davis'; expertise:
'scientist'; salary: 40000; numpublished: 5
%
```

```
printit

UserGlobals at: #bill put: engineer new.
%
printit

bill ssno: '667 52 2261'; name: 'bill johnson'; expertise:
'engineer'; salary: 50000; numprojects: 3
%
```

The messages ssno, name, expertise, salary invoke the respective methods inherited from the EMPLOYEE superclass.

Overriding and Polymorphism

The raise method in the EMPLOYEE superclass defines the annual raise for each object. Scientists receive an annual raise
supplement which is 1% of the annual salary per paper published. Consequently, to implement the raise method for scientist we may reuse the EMPLOYEE raise method for the subclass SCIENTIST.

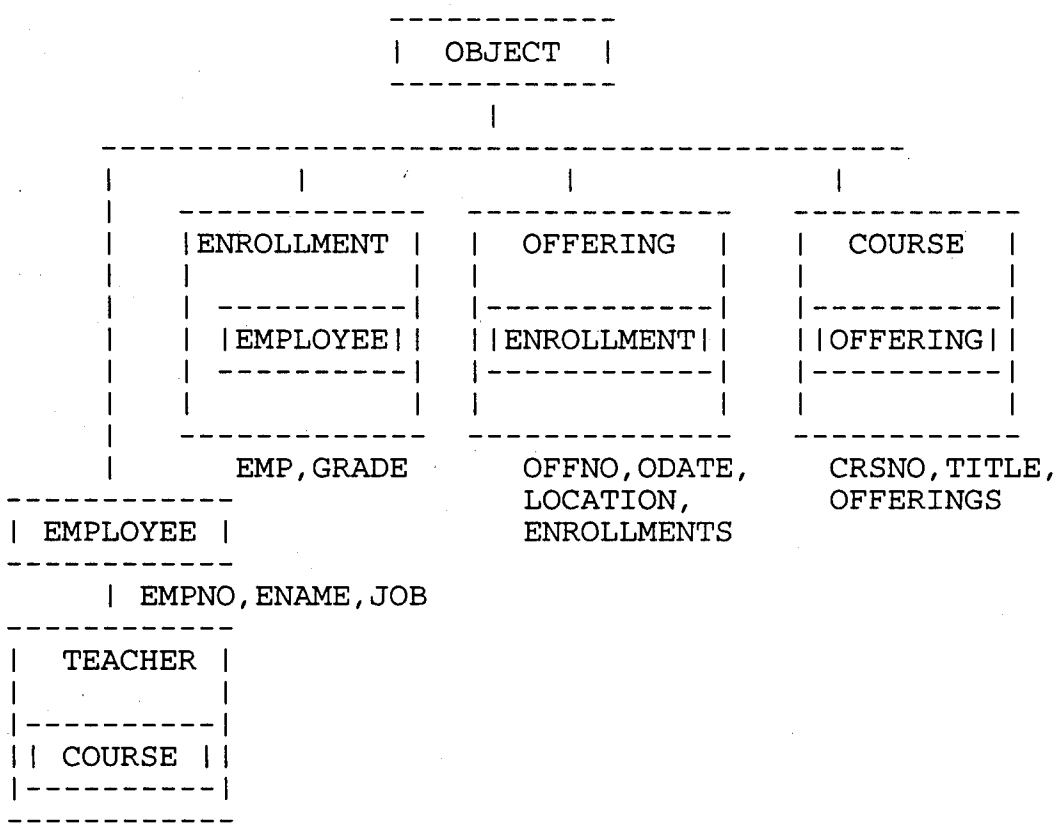
```
set class scientist
method: ^
raise
     ^(super raise) + (salary*numpublished/100)
%
```

Appendix C

Interclass Relationship Implementation

An Education Database

The sequence in which classes of an object model may be defined for an object DBMS is driven by the relationships in the model. We will use C.J. Date's (1995) education enterprise to illustrate the process. Also, writing methods to populate an object database that has relationships is a rather delicate task. Thus after data definition, programs which illustrate the process of populating a database in the presence of interclass relationships are presented. In the process, the typical query capability of an object DBMS is shown. First we derive an object model, Figure C1, for the education enterprise. The education database contains information about an inhouse company education training scheme. The informal statement of the underlying semantics is that for each training course, the database contains details of all offerings of that course; for each offering it contains details of all student enrollments for that offering. The database also contains information about the employees.



COURSES

Figure C1. Object model for an education database

Gemstone Data Definition

The ADT used in a constraint definition must be defined beforehand. Thus, in the definition of a set subclass, the class which constrains the set subclass must be defined first. For this reason, in the example that follows, the 'employee' class is defined before the 'eset' subclass. Also, when an ADT constrains an instance variable, the ADT must be defined first. For example, in the definition of 'enrollment' where 'emp' is constrained by the 'employee' ADT, the 'employee' ADT is defined before 'enrollment'.

"Begin class structure definitions"

"In addition to the five classes in the object model, five collection classes will be defined: employee set, eset; enrollment set, nset; offering set, oset; course set, cset; teacher set, tset."

```
printit
Object subclass: 'employee'
        instVarNames: #('empno' 'ename' 'job')
        inDictionary: UserGlobals
        constraints: #[
                        #[#empno, String],
                        #[#ename, String],
                        #[#job, String]
                         ].
%
```

"Now that the employee class is defined we can define 'eset' which is constrained by employee"

```
printit
Set subclass: 'eset'
        instVarNames: #()
        inDictionary: UserGlobals
        constraints: employee.
%
```

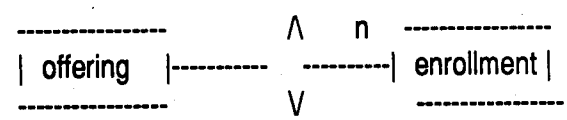
"Enrollment's instance variable 'emp' is also constrained by the already defined 'employee' ADT. Relationship diagram:


```
"
printit
Object subclass: 'enrollment'
        instVarNames: #('emp' 'grade')
        inDictionary: UserGlobals
        constraints: #[
                        #[#emp, employee],
                        #[#grade, String]
                         ].
%
```

"Now that the enrollment class is defined we can define 'nset' which is constrained by enrollment"

```
printit
Set subclass: 'nset'
        instVarNames: #()
        inDictionary: UserGlobals
        constraints: enrollment.
%
```

"Now that 'nset' is defined we can define 'offering' whose instance variable 'enrollments' is constrained by nset. Relationship diagram




```
"
printit
Object subclass: 'offering'
        instVarNames: #('offno' 'odate' 'location' 'enrollments')
        inDictionary: UserGlobals
        constraints: #[
                        #[#offno, String],
                        #[#odate, String],
                        #[#location, String],
                        #[#enrollments, nset]
                         ].
%
```

"Now that 'offering' is defined we can define 'oset' which is constrained by offering"

```
printit
Set subclass: 'oset'
        instVarNames: #()
        inDictionary: UserGlobals
        constraints: offering.
%
```

"Now that 'oset' is defined we can define 'course' whose instance variable 'offerings' is constrained by oset. Relationship diagram

```
----------------          Λ     n    ----------------
|   course   |----------    ----------|  offering   |
----------------          V           ----------------
                           oset
"
```

```
printit
Object subclass: 'course'
        instVarNames: #('crsno' 'title' 'offerings')
        inDictionary: UserGlobals
        constraints: #[
                        #[#crsno, String],
                        #[#title, String],
                        #[#offerings, oset]
                         ].
%
```

"Now that 'course' is defined we can define 'cset' which is constrained by course"

```
printit
Set subclass: 'cset'
        instVarNames: #()
        inDictionary: UserGlobals
        constraints: course.
%
```

"Now that 'cset' is defined we can define 'teacher' whose instance variable 'courses' is constrained by cset. Relationship diagram

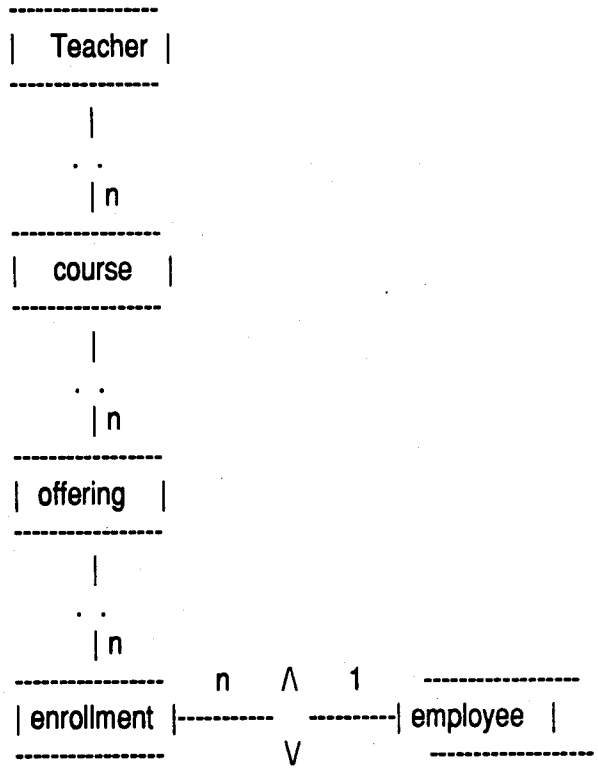
```
----------------          Λ     n    ----------------
|   Teacher  |----------    ----------|   course   |
----------------          V           ----------------
```

```
                        cset
"
printit
employee subclass: 'teacher'
        instVarNames: #('courses')
        inDictionary: UserGlobals
        constraints: #[
                        #[#courses, cset]
                         ]
%
```

"Now that 'teacher' is defined we can define 'tset' which is constrained by teacher"

```
printit
Set subclass: 'tset'
        instVarNames: #()
        inDictionary: UserGlobals
        constraints: teacher.
%
```

Interclass Relationships in the Education Database



Populating/Updating and Retrieving

"Populating the employee class. The Topaz tool makes you set the class for which a method is to be defined"

"The next 3 methods define the methods for initializing/updating this class's instances"

```
set class employee

method: ^
empno: aString
        empno := aString
%
method: ^
ename: aString
        ename := aString
%
method: ^
job: aString
        job := aString
%
```

"The next 3 methods define the methods for retrieving this class's instance values"

```
method: ^
empno
        ^empno
%
method: ^
ename
        ^ename
%
method: ^
job
        ^job
%

set class eset
```

```
"The method, add_empno: add_ename: add_job:, is invoked from the 'set of all employees' to
create a new employee, initialize its instance variables, and include it into the 'set of all
employees'."

method: ^
add_empno: aString1 add_ename: aString2 add_job: aString3

| emp_oid |

"emp_oid = an employee object"

emp_oid := employee new.
emp_oid empno: aString1; ename: aString2; job: aString3.
UserGlobals at: #emp_oid put: emp_oid.

self add: emp_oid.
%

"Create the 'set of all employees'."

printit
UserGlobals at: #SetOfAllEmps put: eset new
%

"Invoke the method, add_empno: add_ename: add_job:"

printit
SetOfAllEmps add_empno:'e009' add_ename: 'watt' add_job: 'engineer'.
%

"Populating the course class. "

set class course

"The next 3 methods define the methods for initializing/updating this class's instances"

method: ^
crsno: aString
        crsno := aString
%
method: ^
title: aString
        title := aString
%
```

```
method: ^
offerings: aString
        offerings := aString
%

"The next 3 methods define the methods for retrieving this class's values"

method: ^
crsno
        ^crsno
%
method: ^
title
        ^title
%
method: ^
offerings
        ^offerings
%

set class cset

"The method add_crsno: add_title:  is invoked from the 'set of all courses' to create a new
course, initialize its instance variables, and include it into the 'set of all courses'."

method: ^
add_crsno: aString1 add_title: aString2

| crs_oid SetOfOfrs |

"crs_oid = a course object
        SetOfOfrs = set of offerings of this particular course"

crs_oid := course new.
crs_oid crsno: aString1; title: aString2.
UserGlobals at: #crs_oid put: crs_oid.

self add: crs_oid.

SetOfOfrs := oset new.
crs_oid offerings: SetOfOfrs.
UserGlobals at: #SetOfOfrs put: SetOfOfrs.
%
"Create a new set of all courses"
```

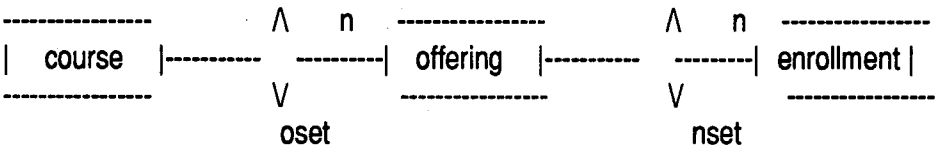
```
printit
UserGlobals at: #SetOfAllCrs put: cset new
%

Invoke the method add_crsno: add_title:

printit
SetOfAllCrs add_crsno: '422' add_title: 'database technology'.
%

"Populating the Offering class. "

set class offering

"The next 4 methods define the methods for initializing/updating this class's instances"

method: ^
offno: aString
        offno := aString
%
method: ^
odate: aDatetime
        odate := aDatetime
%
method: ^
location: aString
        location := aString
%
method: ^
enrollments: aString
        enrollments := aString
%

"The next 4 methods define the methods for retrieving this class's instances"
method: ^
offno
        ^offno
%
method: ^
odate
        ^odate
%
method: ^
```

```
location
        ^location
%
method: ^
enrollments
        ^enrollments
%

set class oset

"The method add_offno: add_odate: add_location: fnd_crs: is invoked from the 'set of all
offerings' to create a new offering, initialize its instance variables, and include it into the 'set of all
offerings'."

method: ^
add_offno: aString1 add_odate: aString2 add_location: aString3 fnd_crs: aString4

| off_oid SetOfEnr crsofr offeroid |

"off_oid = an offering object.  SetOfEnr = set of enrollments for this particular
offering"

off_oid := offering new.

off_oid offno: aString1; odate: aString2; location: aString3.
UserGlobals at: #off_oid put: off_oid.

self add: off_oid.

SetOfEnr := nset new.
off_oid enrollments: SetOfEnr.
UserGlobals at: #SetOfEnr put: SetOfEnr.

"Given a crsno find the corresponding course object, crsofr,  for the new offering object, locate
the set of offerings for the course, offeroid, and add its OID to it

----------------          /\     n    ----------------
|   course    |----------   ----------|  offering    |
----------------          \/          ----------------
                           oset
"

crsofr := SetOfAllCrs detect: [ :cx | aString4 = cx crsno ].
offeroid := crsofr offerings.
```

```
offeroid add: off_oid.
%

"Create the set of all offerings"

printit
UserGlobals at: #SetOfAllOffs put: oset new
%

"Invoke the method add_offno: add_odate: add_location: fnd_crs:"

printit
SetOfAllOffs add_offno:'004' add_odate: '01/8/1996' add_location: 'JAP' fnd_crs:'422'.
%

"Populating the enrollment class. "

set class enrollment

"The next 2 methods define the methods for initializing/updating this class's instances"

method: ^
grade: aString
        grade := aString
%
method: ^
emp: aString
        emp := aString
%

"The next 2 methods define the methods for retrieving this class's instances"

method: ^
grade
        ^grade
%
method: ^
emp
        ^emp
%

set class nset
```

"The method add_grade: fnd_off1: fnd_off2: fnd_emp: is invoked from the 'set of all enrollments' to create a new enrollment, initialize its instance variables, and include it into the 'set of all enrollments'."

```
method: ^
add_grade: aString1 fnd_off1: aString2 fnd_off2: aString3 fnd_emp: aString4

| enr_oid crsofr offeroid ofrenr enroloid empofr |

"enr_oid = an enrollment object"

enr_oid := enrollment new.

enr_oid grade: aString1.
UserGlobals at: #enr_oid put: enr_oid.

self add: enr_oid.

"Given an empno find relevant employee object for this enrollment, empofr,  to determine
existence"

empofr := SetOfAllEmps detect: [ :ex | aString4 = ex empno ].
enr_oid emp: empofr.

"Given a crsno find corresponding course object, crsofr, and given an offno find the relevant
offering object, ofrenr, for this new enrollment object."
```




```
crsofr := SetOfAllCrs detect: [:cx | aString2 = cx crsno ].
offeroid := crsofr offerings.
ofrenr := offeroid detect: [ :ox | aString3 = ox offno ].

"enroloid = set of OIDs of enrollment objects"

enroloid := ofrenr enrollments.
enroloid add: enr_oid.
%

"Create the set of all enrollments"

printit
```

```
UserGlobals at: #SetOfAllEnrs put: nset new
%

"Invoke the method add_grade: fnd_off1: fnd_off2: fnd_emp:"

printit
SetOfAllEnrs add_grade:'a' fnd_off1: '422' fnd_off2: '004' fnd_emp:'e009'.
%

"Populating the Teacher class. "

set class teacher

"The next method defines the methods for initializing/updating this class's instances"

method: ^
courses: aString
        courses := aString
%

"The next method defines the methods for retrieving this class's instances"

method: ^
courses
        ^courses
%

set class tset

"The method fnd_emp: is invoked from the 'set of all teachers' to create a new teacher, initialize
its instance variables, and include it into the 'set of all teachers'."

method: ^
fnd_emp: aString1

| tch_oid emptch |

"tch_oid = a teacher object"

tch_oid := teacher new.

UserGlobals at: #tch_oid put: tch_oid.
```

```
self add: tch_oid.

"Given an empno find employee object, emptch, corresponding to this teacher"

emptch := SetOfAllEmps detect: [ :ex | aString1 = ex empno ].

"Convert the found employee into the new teacher object using substitution"

tch_oid empno: emptch empno.
tch_oid ename: emptch ename.
tch_oid job: emptch job.
%

"Create the set of all teachers"

printit
UserGlobals at: #SetOfAllTchs put: tset new
%

"Invoke the method fnd_emp"

printit
SetOfAllTchs fnd_emp:'e009'.
%
```

Appendix D

Course Outline

Instructor: Kofi Apenyo

JULY 10, 1995 MONDAY 10:00AM - 11:30AM

1. INTRODUCTION
   - Introduction
   - Object
     - Structure
     - Behavior
     - Interaction
     - Definition
     - Object Identifier
   - Class
     - Analogy with Traditional Data Management Terms

2. OBJECT STRUCTURE
   - Structural Abstraction - Instance Variable
     - Composite Instance Variable
     - Object state
     - Class, Instance Variables, and Instances
     - Class variables
     - Constraints
     - Gemstone Class Definition
   - Abstract Data Type
   - Instance Variable Encapsulation
   - Class Hierarchy and "is a"
     - Generalization or Superclass Construction
     - Specialization or Subclass Construction
   - Structural Inheritance
   - Instance Variable Overriding

JULY 11, 1995 TUESDAY 10:00AM - 11:30AM

3. OBJECT METHOD
   - Behavior - Method
   - Kinds of Methods
     - Instance Method
     - Class Method
   - Inheritance of Methods
   - Method Inheritance Algorithm
   - Method Overriding
   - Polymorphism and Dynamic Binding
   - Message

JULY 13, 1995 THURSDAY 10:00AM - 11:30AM

4. OBJECT DBMS

Introduction to Gemstone
A Perception of Instance Variables, Instances, Methods
Protocol
Encapsulation
Demo - Employee database example

JULY 19, 1995 WEDNESDAY 10:00AM - 11:30AM

5. OBJECT INTERACTIONS
   Hierarchy
      The Gemstone Class Hierarchy
   Interclass Relationship
      Connectivity
      Membership
      Aggregation
      Mapping an ER Model to an Object Diagram
      Object Instance Diagram
         Case 1 - Object Diagram for 1:n
         Case 2 - Object Diagram for m:n
   Aggregation Relationship

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	May 1996	Final report

4. TITLE AND SUBTITLE
Object Database Systems: A Tutorial

5. FUNDING NUMBERS

6. AUTHOR(S)
Kofi Apenyo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Engineer Waterways Experiment Station
3909 Halls Ferry Road, Vicksburg, MS 39180-6199

8. PERFORMING ORGANIZATION REPORT NUMBER
Technical Report ITL-96-1

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
U.S. Army Corps of Engineers
Washington, DC 20314-1000

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES
Available from National Technical Information Service, 5285 Port Royal Road, Springfield, VA 22161.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

An object model and the corresponding instance diagram embody all the concepts-object identity, structure, behavior, interaction, inheritance, and encapsulation-of the object-oriented paradigm. In an object-oriented data model, an entity is represented as an instance (object) of a class that has a set of properties and operations (methods) applied to the objects. A class, and hence an object, may inherit properties and methods from related classes. Objects and classes are dynamic and can be created at any time. With an object model, database implementation may begin.

As with many other areas in computer science, true understanding of a novel concept comes only from hands-on work. In the case of object-oriented database systems, this means constructing simple object models and writing programs to implement and process them. This tutorial gives the fundamentals to tackle these tasks.

14. SUBJECT TERMS
Computer-aided design
Computer-aided software engineering
Database administrator
Database management system
Geographic information system
Object identifier
Object oriented

15. NUMBER OF PAGES
126

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED		

Destroy this report when no longer needed. Do not return it to the originator.